



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

2004-12

Building software tools for combat modeling and analysis

Chen, Yuanxin

Monterey, California. Naval Postgraduate School

<http://hdl.handle.net/10945/1282>

Copyright is reserved by the copyright owner

Downloaded from NPS Archive: Calhoun



<http://www.nps.edu/library>

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**BUILDING SOFTWARE TOOLS FOR COMBAT
MODELING AND ANALYSIS**

by

Chen Yuanxin

December 2004

Thesis Advisor:
Second Reader:

Mikhail Auguston
Richard Riehle

Approved for public release: Distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE December 2004	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE: Building Software Tools for Combat Modeling and Analysis		5. FUNDING NUMBERS	
6. AUTHOR(S) Chen Yuanxin			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release: Distribution is unlimited		12b. DISTRIBUTION CODE	
13. ABSTRACT (<i>maximum 200 words</i>) The focus of this thesis is to use and leverage the strengths of dynamic computer program analysis methodologies in software engineering testing and debugging such as program behavior modeling and event grammars to automate the building and analysis of combat simulations. An original high level language METALS (Meta-Language for Combat Simulations) and its associated parser and C++ code generator were designed to reduce the amount of time and developmental efforts needed to build sophisticated real world combat simulations. A C++ simulation of the Navy's current mine avoidance problem in littoral waters was generated using high level METALS description in the thesis as a demonstration. The software tools that were developed will allow users to focus their attention and efforts in the problem domain while sparing them to a considerable extent the rigors of detailed implementation.			
14. SUBJECT TERMS Event Grammar, Context Free Grammar, BNF, Rigal, Lexical Analyzer, Language Parser, Code Generator, Mine Avoidance Concept, METALS, Latvia			15. NUMBER OF PAGES 201
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release: distribution is unlimited

BUILDING SOFTWARE TOOLS FOR COMBAT MODELING AND ANALYSIS

Chen Yuanxin
Major, Republic of Singapore Navy
B.S. (Hons) Elect, Nanyang Technological University, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
December 2004**

Author: Chen Yuanxin

Approved by: Dr. Mikhail Auguston
Thesis Advisor

Dr. Richard Riehle
Second Reader

Dr. Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The focus of this thesis is to use and leverage the strengths of dynamic computer program analysis methodologies in software engineering testing and debugging such as program behavior modeling and event grammars to automate the building and analysis of combat simulations.

An original high level language METALS (Meta-Language for Combat Simulations) and its associated parser and C++ code generator were designed to reduce the amount of time and developmental efforts needed to build sophisticated real world combat simulations. A C++ simulation of the Navy's current mine avoidance problem in littoral waters was generated using high level METALS description in the thesis as a demonstration. The software tools that were developed will allow users to focus their attention and efforts in the problem domain while sparing them to a considerable extent the rigors of detailed implementation.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	BACKGROUND	1
B.	THE PROCESS OF BUILDING COMBAT MODELS BASED ON THE DYANMIC ANALYSIS APPROACH	4
C.	OTHER SIMILAR OR RELATED WORK IN THIS FIELD	6
II.	THE EVENT BASED METALANGUAGE FOR SIMULATIONS	9
A.	LANGUAGE DESIGN OVERVIEW	9
B.	STEP 1 - PROGRAM BEHAVIOR MODELING	9
C.	STEP 2 - EVENT MODELING	12
D.	STEP 3 - RELATIONSHIPS BETWEEN EVENTS	15
E.	STEP 4 - METALS SYNTAX AND SEMANTICS	17
III.	IMPLEMENTATION OF METALS	37
A.	AN OVERVIEW OF THE METALS COMPILER	37
B.	COMPILER DEVELOPMENT TOOL	38
C.	THE METALS LEXICAL ANALYZER	45
D.	THE METALS PARSER	46
E.	THE METALS CODE GENERATOR	76
IV.	APPLICATIONS OF METALS	101
A.	EXAMPLE 1 - SIMPLE COIN TOSS	101
B.	EXAMPLE 2 - MINE AVOIDANCE IN LITTORALS	108
V.	CONCLUSION	121
	APPENDIX A - SOURCE CODE FOR METALS COMPILER 118	123
	APPENDIX B - MINE AVOIDANCE SIMULATION IN METALS 136	143

APPENDIX C - MINE AVOIDANCE SIMULATION IN GENERATED C++	161
LIST OF REFERENCES	179
INITIAL DISTRIBUTION LIST	181

LIST OF FIGURES

Figure 1.	Simulating Discrete Events.....	2
Figure 2.	Incorporating continuous variables into discrete events simulations.....	3
Figure 3.	Overview of the Combat Model Building Process.....	5
Figure 4.	The role of the Meta Language in simulation.....	10
Figure 5.	An Event Set.....	15
Figure 6.	An Event Chain.....	15
Figure 7.	Alternative Events.....	16
Figure 8.	Concurrent Events.....	17
Figure 9.	Code example for Rules 2 and 3.....	20
Figure 10.	Code example for Rule 4.....	21
Figure 11.	Code example for Rule 5.....	22
Figure 12.	Code example for Rule 6.....	24
Figure 13.	Code example for Rule 7.....	25
Figure 14.	An Event Chain - Part 2.....	26
Figure 15.	An Event Chain - Part 3.....	26
Figure 16.	An Event Chain - Part 3.....	27
Figure 17.	The SIMPLE Pattern.....	28
Figure 18.	The ACTION Pattern.....	29
Figure 19.	The CONDITIONAL Pattern.....	30
Figure 20.	The ALTERNATIVE Pattern - Part 1.....	30
Figure 21.	The ALTERNATIVE Pattern - Part 2.....	31
Figure 22.	The ALTERNATIVE Pattern - Part 2.....	31
Figure 23.	Special case for ALTERNATIVE Pattern - Part 1A.....	32
Figure 24.	Special case for ALTERNATIVE Pattern - Part 1B.....	32
Figure 25.	Special case for ALTERNATIVE Pattern - Part 2A.....	32
Figure 26.	Special case for ALTERNATIVE Pattern - Part 2B.....	32
Figure 27.	The ALTERNATIVE Pattern.....	33
Figure 28.	The ITERATION Pattern - Part 1.....	34
Figure 29.	The ITERATION Pattern - Part 2.....	34
Figure 30.	The ITERATION Pattern - Part 3.....	34
Figure 31.	The GROUP Pattern.....	35
Figure 32.	Composite Events.....	35
Figure 33.	Components of the METALS Compiler.....	38
Figure 34.	A Simple RIGAL Program.....	39
Figure 35.	A Simple List Comprising 3 Atoms.....	40
Figure 36.	A Simple Tree Comprising 2 Atoms.....	41
Figure 37.	A Multi-layer Tree Comprising 3 Atoms.....	42
Figure 38.	A Multi-layer Tree Comprising 5 Atoms.....	42
Figure 39.	Tree Addition.....	43
Figure 40.	A Simple RIGAL Rule.....	44
Figure 41.	An example of an input stream of tokens.....	45
Figure 42.	An example of code with syntax error.....	45
Figure 43.	Input source code.....	46
Figure 44.	Output list of tokens.....	46

Figure 45.	Expected Input Pattern for a METAL programs.....	47
Figure 46.	Program Execution Flow for #Parse Rule.	47
Figure 47.	The METALS Parsed Tree Structure.	48
Figure 48.	The METALS Parsed Tree.	48
Figure 49.	Expected Input Pattern for #Title	49
Figure 50.	Program Execution Flow for #Title Rule.....	49
Figure 51.	Output returned by the #Title rule.	49
Figure 52.	The #Title Parsed Tree Structure.....	49
Figure 53.	Expected Input Pattern for #Title	50
Figure 54.	Output returned by the #Title rule.	50
Figure 55.	The #Header Structure.....	50
Figure 56.	Expected Input Pattern for #World	51
Figure 57.	Program Execution Flow for #World Rule.....	51
Figure 58.	Output returned by the #World rule.....	52
Figure 59.	The #World Parsed Tree Structure.....	52
Figure 60.	Expected Input Pattern for #Entity	53
Figure 61.	Program Execution Flow for #Entity Rule.	53
Figure 62.	Output returned by the #World rule.....	53
Figure 63.	The #Entity Parsed Tree Structure.	54
Figure 64.	Expected Input Pattern for #Event	54
Figure 65.	Program Execution Flow for #Event Rule.....	55
Figure 66.	Output returned by the #Event Rule.	55
Figure 67.	The #Event Parsed Tree Structure.....	55
Figure 68.	Expected Input Pattern for #Event_Attributes	56
Figure 69.	Program Execution Flow for #Event_Attributes Rule.	56
Figure 70.	Output returned by the #Event_Attributes rule.	56
Figure 71.	The #Event_Attributes Parsed Tree Structure.	57
Figure 72.	Expected Input Pattern for #Event	57
Figure 73.	Program Execution Flow for #Chain rule.....	58
Figure 74.	Output returned by the #Chain rule.....	58
Figure 75.	The #Chain Parsed Tree Structure.....	58
Figure 76.	Expected Input Pattern for #Rule	59
Figure 77.	Program Execution Flow for #Rule rule.	59
Figure 78.	Output returned by the #Chain rule.....	59
Figure 79.	The #Rule Parsed Tree Structure.	60
Figure 80.	The Expected Input Pattern for #Pattern	61
Figure 81.	Output for a single event pattern.....	61
Figure 82.	Output for a list of event patterns.	62
Figure 83.	The #Pattern Parsed Tree Structure for A Single Pattern.....	62
Figure 84.	The #Pattern Parsed Tree Structure for Lists of Patterns.	62
Figure 85.	Expected Input Pattern for #Iteration	63
Figure 86.	Output returned by the #Iteration rule.....	63
Figure 87.	The #Iteration Parsed Tree Structure.....	64
Figure 88.	Expected Input Pattern for #Loop	64

Figure 89.	Output returned by the #Loop rule.	65
Figure 90.	The #Loop Parsed Tree Structure.	65
Figure 91.	Expected Input Pattern for #Conditional .	65
Figure 92.	Output returned by the #Conditional rule.	66
Figure 93.	The #Conditional Parsed Tree Structure.	66
Figure 94.	Expected Input Pattern for #Conditional .	66
Figure 95.	Output returned by the #Alternative rule.	67
Figure 96.	Expected Input Pattern for #Outcome .	67
Figure 97.	Expected Input Pattern for #Outcome .	68
Figure 98.	Output returned by the #Probability rule.	68
Figure 99.	Output returned by the #Outcome rule.	68
Figure 100.	Output returned by the #Alternative rule.	68
Figure 101.	The #Alternative Parsed Tree Structure.	69
Figure 102.	Expected Input Pattern for #Simple .	69
Figure 103.	Output returned by the #Simple rule.	70
Figure 104.	The #Simple Parsed Tree Structure.	70
Figure 105.	Expected Input Pattern for #Group .	70
Figure 106.	Output returned by the #Alternative rule.	71
Figure 107.	The #Group Parsed Tree Structure.	71
Figure 108.	Expected Input Pattern for #Group .	71
Figure 109.	Output returned by the #Action rule.	72
Figure 110.	The #Action Parsed Tree Structure.	72
Figure 111.	Program Execution Flow for #Pattern .	73
Figure 112.	The METALS Parsed Tree Structure.	76
Figure 113.	Structure of generated C++ simulation program.	76
Figure 114.	Opening 5 output files for writing.	77
Figure 115.	Feeding the parsed tree into specific code generating rules.	77
Figure 116.	Generating the C++ main program.	78
Figure 117.	Input to #Generate_Title Rule.	78
Figure 118.	Output from #Generate_Title Rule.	78
Figure 119.	Input to #Generate_Headers Rule.	79
Figure 120.	Output from #Generate_Headers Rule.	79
Figure 121.	Input to #Generate_Worlds Rule.	80
Figure 122.	Data structure for each world object in list \$Worlds .	80
Figure 123.	Output for each world object from #Generate_Worlds Rule in Worlds.h .	81
Figure 124.	Input to #Generate_Entities Rule.	82
Figure 125.	Data structure for each entity object in list \$Entities .	82
Figure 126.	Output from #Generate_Entities Rule in Entities.h .	82
Figure 127.	Input to #Generate_EventClasses Rule.	83
Figure 128.	Data structure for each event object in list \$Events .	83
Figure 129.	Output from #Generate_Events Rule in Events.h .	84
Figure 130.	Event chain code generation mapping.	85
Figure 131.	Input to #Generate_EventChains Rule.	86

Figure 132.	The input data structure of an event chain.....	86
Figure 133.	Code generated directly by #Generate_EventChains	86
Figure 134.	The input data structure of an event chain.....	87
Figure 135.	Code generated directly by #Generate_Rules	87
Figure 136.	The input data structure of an event pattern.....	88
Figure 137.	Mapping of code generating rules to event types.....	88
Figure 138.	Input to #Generate_Iteration Rule.....	89
Figure 139.	Code generated by #Generate_Iteration when \$Op = '='	90
Figure 140.	Portion of the RIGAL source code to implement iteration.....	90
Figure 141.	Code generated by #Generate_Iteration when \$Op is not '='.....	91
Figure 142.	Input to #Generate_Alternative Rule.....	91
Figure 143.	Code generated by #Generate_Probability for n outcomes.....	92
Figure 144.	Portion of the RIGAL source code to implement alternatives.....	92
Figure 145.	Standard print the C++ pseudorandom number generator in the output....	93
Figure 146.	Input to #Generate_Probability Rule.....	95
Figure 147.	Portion of the RIGAL source code to implement IF-ELSE statements....	96
Figure 148.	Input to #Generate_Alternative Rule.....	96
Figure 149.	Code generated by #Generate_Conditional	97
Figure 150.	Input to #Generate_Loop Rule.....	97
Figure 151.	Code generated by #Generate_Loop	97
Figure 152.	Input to #Generate_Simple Rule.....	98
Figure 153.	Code generated directly by #Generate_Simple	98
Figure 154.	Input to #Generate_Alternative Rule.....	99
Figure 155.	METALS source code for Simple Coin Toss example.....	101
Figure 156.	Intermediate parsed tree produced by METALS parser.....	103
Figure 157.	Listings for MAINCPP generated by METALS code generator.....	104
Figure 158.	Listings for ENTITIESH generated by METALS code generator.....	104
Figure 159.	Partial Listings for EVENTSH generated by METALS code generator....	105
Figure 160.	Listings for CHAINSH generated by METALS code generator.....	106
Figure 161.	Screen output by program after execution.....	106
Figure 162.	Mine Clearance State Diagram.....	109
Figure 163.	Mine Avoidance State Diagram.....	110
Figure 164.	Event Trace for Mine Avoidance.....	111
Figure 165.	Entities needed in the simulation.....	112
Figure 166.	Pseudo-code for Event Move.....	113
Figure 167.	METALS source code to set up a minefield.....	114
Figure 168.	METALS source code calculate the expected number of mines and NOMBOs.....	114
Figure 169.	Algorithm for spreading mines and objects.....	115
Figure 170.	Algorithm for setting the object type for each Minefield square.....	115
Figure 171.	Spatial Poisson Process for Mine Avoidance.....	116
Figure 172.	Computing duration over event traces.....	117
Figure 173.	Screen output by Mine Avoidance simulation program after execution 1.	119
Figure 174.	Screen output by Mine Avoidance simulation program after execution 2.	119
Figure 175.	Screen output by Mine Avoidance simulation program after execution 3.	120

LIST OF TABLES

Table 1.	Relationships Between Events.....	16
Table 2.	Token pair concatenation table	75
Table 3.	Iteration Operators	89
Table 4.	Continuous Pseudorandom Variate Generators	94
Table 5.	Discrete Pseudorandom Variate Generators	95
Table 6.	Mine Avoidance Events	111
Table 7.	Mine Avoidance Entities.....	112
Table 8.	Results of Simulation Runs.....	118

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to offer my deepest gratitude to my advisor Dr Mikhail Auguston for his clinical insight, invaluable guidance and excellent facilitation during the course of work on this thesis. His remarkable patience, constant encouragement, faith in my abilities and wonderful ability to teach and explain the abstract and the complicated with great clarity and simplicity has untied countless dead ends and reduced the number of sleepless nights.

I would also like to thank my second reader Dr. Richard Riehle for his assistance and for taking time out to go through my work. His presence and remarks have always been an inspiration and a constant source of motivation.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

As an active area of research in the field of software engineering, dynamic computer program analysis methodologies such as program behavior modeling and event grammars are currently used to automate software analysis, testing and debugging.

In program behavior modeling, a computer program is modeled as a set of inter-related events over finite time (event trace). Each program event is abstracted as an object (event type) with attributes. Relationships such as precedence between events are also defined.

The behavior of events is described formally using context free semantic rules known as event grammars. A specialized high level programming language based on the notion of event grammars can consequently be designed for the purpose of describing program behavior and to perform computations over event traces. Such program behavior modeling is traditionally used in software analysis, testing and debugging, whereby specific routines are built into these precise program behavior models for assertion checking, debugging queries, execution profiles and performance measurements.

This thesis extended this concept in a new direction by using the methodology to build combat simulation models and analyze their behavior. A high level language called "Meta-Language for Combat Simulations" or METALS is designed and implemented, along with its parser and C++ code generator to enable users to describe and build sophisticated combat models with a high level of abstraction and simplicity. Robust and complex combat simulation C++ programs can be created automatically using high level description in METALS.

The language parser and code generator are written using the RIGAL compiler construction language tools developed at the University of Latvia. As an important demonstration in the thesis, METALS was used to model the Navy's real world and doctrinal problem of Mine Avoidance in littoral waters. The stochastic processes and the set of definitive and procedural events that occur are translated from relatively simple high level

definitions by METALS into their equivalent and substantially longer and more complex lower language C++ code. METALS will allow users to focus their attention more on the problem at hand than on the rigors of its implementation in lower level code.

I. INTRODUCTION

A. BACKGROUND

1. The Existing Approach To Software Debugging Automation

The existence of unintended, hidden or unknown behaviors in a computer program can have serious security, reliability, safety and other repercussions for the system(s) within which the program reside. Therefore the dynamic analysis of the behavior of substantial and complex computer programs while they are running constitutes an important and active area of research in software engineering. In dynamic analysis, the specific behaviors of a computer program such as sequences of steps performed, histories of variable values, function call hierarchies, frequencies of traversed paths, frequencies of execution of program blocks (program hot spots), race conditions and memory reference errors are typically instrumented either by inserting instrumentation code (assertions) directly into the source code, using the compiler itself to insert the code during compile time, or through the re-writing of binary images containing the text sections for shared libraries or other applications [AUG03] [BALL99]. Automated tools for performing these often manually performed and time consuming techniques cannot be developed easily without a more precise and formal representation (model) of program behavior. In other words, tools or programs developed to instrument other programs automatically will need to recognize the standard patterns of behavior in the programs and the semantic rules that govern these patterns of behavior. A high level language that can be used to model program behaviors based on events and event traces will therefore be extremely useful in this context. Notable examples of such languages that have been designed for debugging automation include the EEL [LARUS95], FORMAN [AUG98], UFO [AUG03].

2. Adapting The Approach To Build Combat Simulation Models

A combat simulation model essentially simulates an abstract representation of a particular military system. A military system typically comprises a set of entities such as platforms, weapons, sensors and soldiers mutually interacting and consuming resources

within a physical environment resulting in a chain of co-related events. Therefore simulation models of military systems are frequently discrete-event simulation models where the state of the system only changes at discrete points of time, and a change in state is driven by an event (event-driven). In additional, each event also exists autonomously and is discrete, implying that nothing happens during the transition between one to the next.

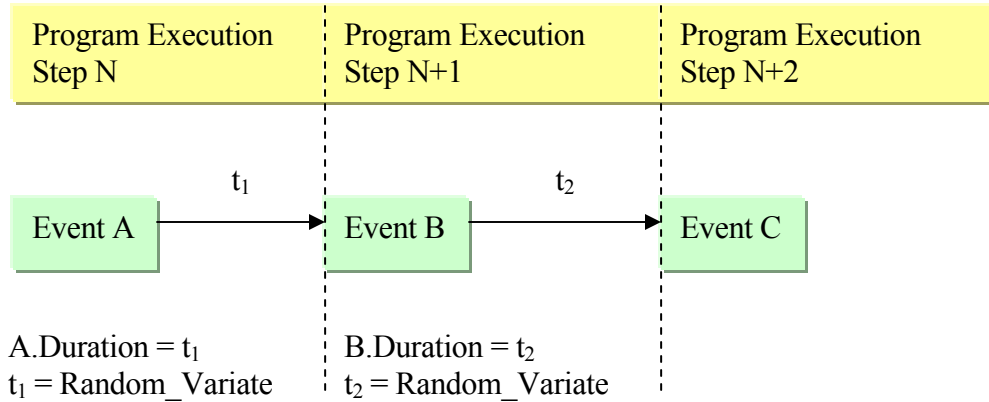


Figure 1. Simulating Discrete Events.

As illustrated in Figure 1 above, discrete events are often simulated in a computer program by treating each program execution step (each cycle in an iteration or loop in terms of data structure) to represent a transition between successive events. The stochastic nature of the intervals between two events can be implemented by defining a duration attribute for each event and then updating this attribute with a randomly generated value around some empirically obtained mean value during a transition.

A continuous-event simulation on the other hand is characterized by having the state of the system changes constantly over time (time-driven). Although this thesis's focus is primarily on simulation of discrete event systems, an event driven simulation model can also be extended to incorporate continuous variables by dividing the randomly generated event duration into quasi-fixed time intervals, and to perform functional computations on continuous variables over these quasi-fixed time intervals using as illustrated in figure 2 below.

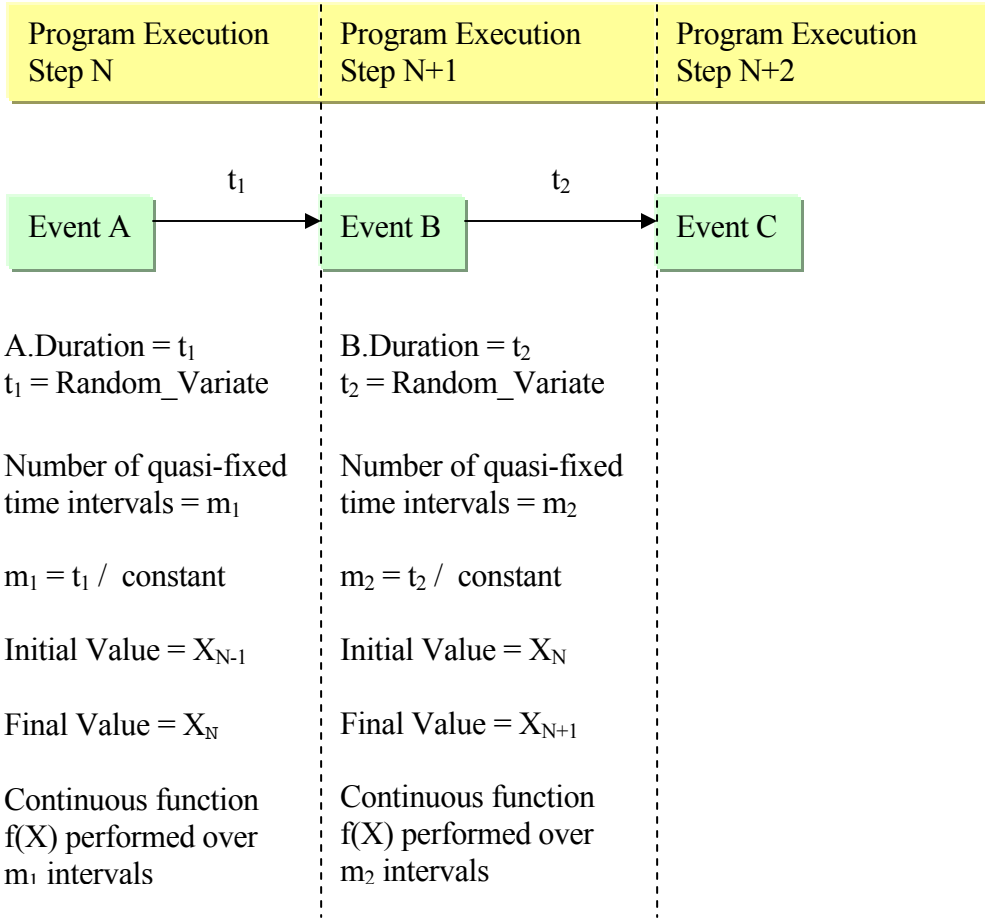


Figure 2. Incorporating continuous variables into discrete events simulations.

Discrete-event simulation models are usually analyzed numerically and exist essentially as computer programs in slight contrast with other types of models such as mathematical (analytic), statistical and input-output models. Because discrete-event systems are built from the basic notion of an event as an elementary unit of action, the dynamic analysis program behavior modeling approach described in the previous section can be adapted to model and describe the behavior of discrete-event military systems. This forms the focus of this thesis research. Instead of automating the generation computer programs with instrumentation code and functionalities built-into them using some high level language, the main aim here is to design and develop an entirely new high level language centered around the notion of events to built or generate discrete-event combat simulation models automatically in lower level code. In this thesis, the conception, design and implementation of the high level language called the Meta Language for Combat

Simulations (METALS) will be described in detail. Such a high level language will allow users to focus their attention more on the simulation problem at hand than on the rigors of its implementation in lower level code.

B. THE PROCESS OF BUILDING COMBAT MODELS BASED ON THE DYNAMIC ANALYSIS APPROACH

A simulation program can be written directly in any native general purpose language such as FORTRAN, Basic, C, C++ or Java. However for practical reasons, a discrete event simulation computer program will need to be written in a language that best supports the abstraction of real world entities, interactions and processes. Object oriented languages such as C++ or Java will therefore be the two most suitable candidates, due to their widespread usage and the availability of development tools and compilers. The aim of the thesis is not to design an entirely new language for simulation that duplicates the full functionalities of these languages, the purpose but to automate the generation of programs written more robustly in these languages by allowing users to model a real world system using intuitive and higher level descriptions and definitions. In this thesis, the chosen language for the generated simulation program is C++ on a PC as a matter of preference, although the entire prescribed methodology will technically work with any target language.

The process of building combat models based on the dynamic analysis approach is envisaged to comprise of several intermediate stages as illustrated in Figure 3.

In Stage 1, a high level description of the model is first created by the user visually using a graphical user interface engine. The output of this process is a text file containing the description of the model in the high level language.

In Stage 2, a custom built compiler comprising a language parser and code generator will be involved to translate the high level model description into the equivalent C++ program code.

In Stage 3, a C++ compiler will compile the equivalent C++ code into an executable

that is ready to be run on the targeted platform.

Stages 4 and 5 are highly automated tasks involving extensive simulation runs and data aggregation and analysis.

The main focus and most important research work accomplished and described in this thesis is the design and development of the high level language needed in Stage 1 as well as its associated language parser and code generator in Stage 2. A detailed description of this high level language - the Meta-Programming Language for Simulation or METALS and the implementation details of the parser and code generator will be presented in Chapters I and III respectively.

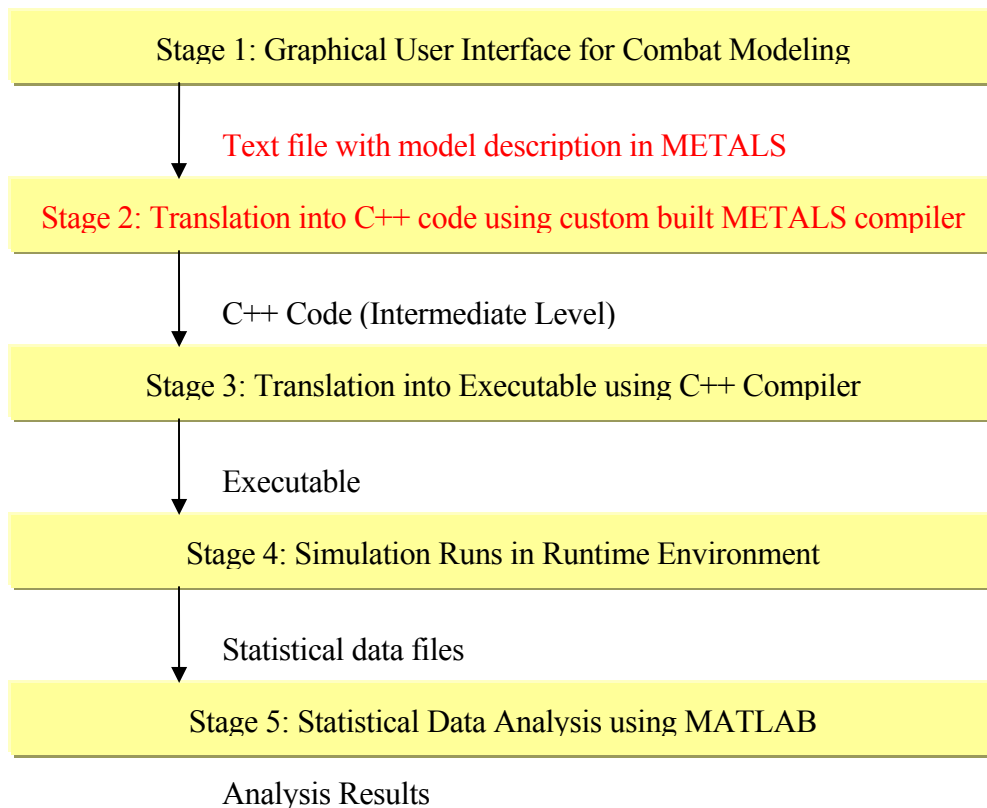


Figure 3. Overview of the Combat Model Building Process.

C. OTHER SIMILAR OR RELATED WORK IN THIS FIELD

1. Early Programming Languages Designed Specifically For Simulation

High level object-orientated simulation programming first originated with the SIMULA language [NYGAARD] first proposed in 1966. Although SIMULA can be used as general purpose language, it was designed to address the problem of describing complex real world systems in discrete simulations. It introduced the record class as a form of data abstraction and is a precursor to object-oriented programming languages like Smalltalk and C++.

In the simulation of continuous systems, the Continuous System Simulation Language (CSSL) [CSSL67] was first developed by the Simulations Council Inc (SCI) for the Jet Propulsion Lab and became commercially available in 1968, and was widely distributed from 1969-1975 [COMDIC1]. A later variant of CSSL, the Continuous System Modeling Program (CSMP) [SPECK76] was first published in 1976. IBM developed a APL compiler for simulations written in the CSMP language in the late 1970s and introduced three different products based on this compiler in the mainframe computers of the time.[IBM79]. In the CSMP language, models are directly entered in the form of mathematical equations and the language provides a large number of predefined mathematical and engineering functions and blocks [ALFON99].

The earliest simulation programming language based on the notions of Entities, Attributes, Sets and Events (EAS-E) was the original SIMSCRIPT [I] language developed by Harry Markowitz et al at the RAND Corporation in Santa Monica [EAS-E][HYPER1]. As a precursor to FORTRAN, the language with a free-form English like grammar influenced SIMULA and was designed primarily as a language for programming discrete events. The latest incarnation of the language, the SIMSCRIPT II.5 is developed, maintained and currently marketed as a complete but proprietary and expensive simulation development environment by CACI.

2. Recent Meta Programs And Languages Used To Generate Simulation Programs In A Lower Level Language

Meta programs are programs used to generate another program usually in a lower level languages. The most common examples of meta programs languages are the macro pre-processors, compilers and interpretators.

A more recent example of a meta program used to generate simulation programs is the APL CSMP/OO-CSMP compiler developed by Manuel Alfonseca et al [ALFON99] in 1999. His team first developed a compiler written in APL to translate simulation models written in the CSMP language into APL simulation programs. His team also extended the CSMP language to create the new Object Oriented CSMP (OO-CSMP) which incorporated object oriented features such as definition of objects and classes, simple inheritance, vectors of objects, definition of attributes and functions (methods) associated to a class of objects and a simple way to reference object and object vectors attributes and methods. They developed a separate compiler written in APL2 to translate simulation models written in OO-CSMP into C++ simulation programs. C++ was chosen to overcome performance problems the team experience with APL programs. Both the APL and APL2 compilers comprised of a lexical analyzer, a syntax analyzer (parser) and a code generator.

A metalanguage is a formal language used to describe other object languages. Event based metalanguages such as EEL [LARUS95], FORMAN [AUG98] and UFO [AUG03] are used to define and insert instrumentation code inside a target object language like C++. For simulation model construction, the OO-CSMP language described earlier is suitable for modeling continuous systems and the time based interactions between entities inside these systems. The event based metalanguage for combat simulations (METALS) presented in this thesis relatively unique in this regard.

THIS PAGE INTENTIONALLY LEFT BLANK

II. THE EVENT BASED METALANGUAGE FOR SIMULATIONS (METALS)

A. LANGUAGE DESIGN OVERVIEW

The design of the metalanguage METALS begins with the rationalization and aggregation of the behavior of a simulation program. There are four important steps that are integral to this process of rationalization, namely:

- Step 1 - Differentiating between the design and problem domain behaviors of the program.
- Step 2 - The abstraction of each design domain event and simulated real world event as an object (event type) with attributes and methods.
- Step 3 - The definition of a set of relationships between events over finite time.
- Step 4 - The formal method of translating of system behavior into event traces using event grammars.

B. STEP 1 - PROGRAM BEHAVIOR MODELING

The behavior of a simulation program is first examined in two different contexts, namely in the context of the program's design and flow (design domain) and in the context of the simulated event flows (problem domain).

1. Design Domain Preambles

In the design domain, all event driven simulation programs typically share a common set of characteristics and events. The common characteristics include:

- The ability to perform multiple simulation runs and to allow users to define the initial and terminating conditions for each of these runs.
- The ability of perform statistical computations and performance instrumentation over the simulation runs.
- The provisions of special routines and libraries such as random variable generators to support the simulation.
- The support for the object-oriented modeling of real world entities and events.

The common design domains events are largely dictated by the specific programming language in which the program is developed. For example, a typical sequence of events that occur during the writing of a standard C++ program is illustrated below:

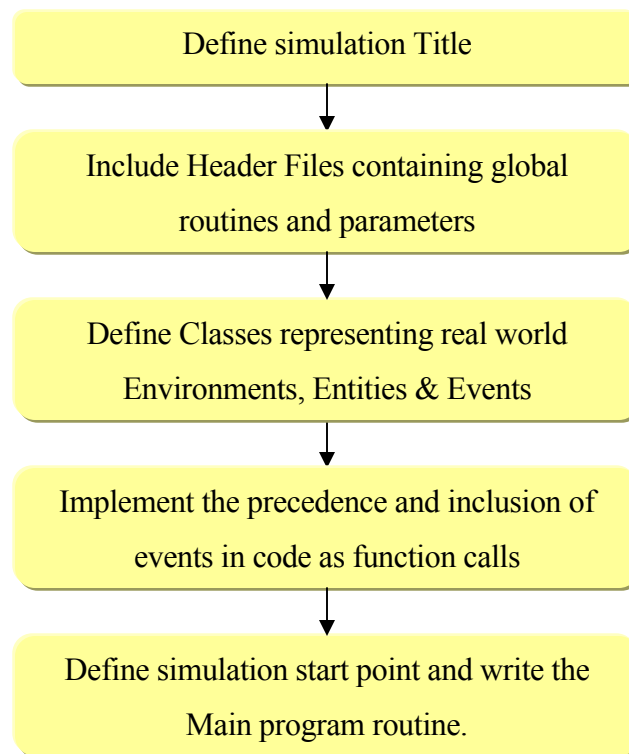


Figure 4. Sequence of Events in C++ Simulation Program Creation.

This means that METALS will need to have a syntax that supports the various types of programming activities albeit from a higher level of abstraction. For example, a preliminary context free language rule based on the sequence of events shown in Figure 4 can be defined as follows:

$$\begin{aligned} \langle \text{Simulation_Program} \rangle &::= \\ \langle \text{Title} \rangle \langle \text{Headers} \rangle \langle \text{Entities} \rangle \langle \text{Events} \rangle \langle \text{Rules} \rangle \end{aligned} \quad (2.1)$$

Rule 2.1 stipulates that a simulation program can be defined as a series of title, headers, entities, events and rules that govern how the entities and events interact. Each of the five non-terminating symbols on the right hand side of the rule can subsequently be further defined. For example, the $\langle \text{Headers} \rangle$ symbol represents a list of headers file that the user intends to include in the program can be represented as an iterative series of similar series header $\langle \text{Header} \rangle$ which in turn is terminated by the reserved word "include" followed by the user specified filename as shown in Rules 2.2 and 2.3 below.

$$\langle \text{Headers} \rangle ::= \langle \text{Header}^* \rangle \quad (2.2)$$

$$\langle \text{Header} \rangle ::= \text{"include"} \langle \text{filename} \rangle \quad (2.3)$$

Here the asterisk symbol in rule 2.2 is a standard BNF (Backus Naur form) symbol used to represent iteration. A complete walk through of the final METAL language in context free grammar BNF will be given in the later sections of this chapter.

2. Problem Domain Preambles

In the problem domain, the basic assumption made in the design of the language is that any military system will comprise entities generating chains of interrelated events. The behavior of the simulated system is then abstracted as a set of inter-related events over finite time (event trace).

The sequence and order in which events occur in the real world can be aggregated along a few standard patterns of behavior, which in turn will be used to construct the syntax and semantics of the METALS for simulation. These standard patterns include:

- Iteration - The same set or sequence of events is repeated a number of times.
- Loop - The same set or sequence of events is repeated while some condition in the system remains true.
- Conditional - A set or sequence of events is set to occur if the condition is right, if not an alternative set or sequence of events will result.
- Alternative - There exist a number of alternative event(s) that can follow a set or sequence of events, each alternative having some probability of occurrence.
- Concurrency - A parallel set or sequence of events in occurring concurrently with the current set or sequence of events.
- Action - Each event typically occurs with some interaction between entities in the system resulting in a manipulation of the system, entity or events attributes.

C. STEP 2 - EVENT MODELING

1. Basic Definition And Characteristics Of Events

An event is defined as something that happens at a given place and time. Every event also has a duration which is based from the event's start and finishing times, and can either be deterministic or stochastic.

Two basic binary relationships exist between events, namely precedence and

inclusion [AUG98].

Any arbitrary event can either precede or be included in another event. The following example shows an event trace for event A. In this simple event trace, event B occurs before C (Precedence) and both events B and C in sequence constitutes a larger event A (Inclusion).

$$\text{Rule A} :: \text{B C} \quad (2.3)$$

The precedence relation defines partial order of events. Two events are not necessary ordered for they also can happen concurrently. The following example illustrates this. In this event trace, events C and D are concurrent events which occurs after event B but before event E.

$$\text{Rule A} :: \text{B } \{ \text{C D} \} \text{E} \quad (2.4)$$

Such an event based approach to modeling real world behavior is compatible with the principles of discrete event modeling and simulation. The completion of one event triggers another. An event is not a point in the state space of the system but has a duration (starting and ending time) within which its attributes can vary. Such a duration can be deterministic (fixed) for a particular class of events or be stochastic in which inter-arrival times between events is randomly distributed about some known mean value or parameters. The manipulation of an event's attributes is also useful, especially in simulating the consumption of resources.

2. Formal Axioms

The formal axioms [AUG98] that should be satisfied by any events based on these two binary relationships include:

- Mutual exclusion

A **PRECEDES** B implies that A is not **IN** B (2.5)

- Non-commutability

A PRECEDES B implies that B does NOT PRECEDES A (2.6)

A IN B implies that B is NOT IN A (2.7)

- Transitivity

If A PRECEDES B, and B PRECEDES C,
then A PRECEDES C (2.8)

- Distributivity

If A is IN B, and B PRECEDES C,
then A PRECEDES C (2.9)

If A PRECEDES B, and B is IN C,
then A PRECEDES C (2.10)

D. STEP 3 - RELATIONSHIPS BETWEEN EVENTS

1. Basic Relationships

The last step prior to the design of the language syntax and semantics is to formalize the relationships between events.

- **Event Set** - A set of events or event set is expressed as an event that is composed of a set of other unrelated or concurrent events as shown in Figure 5 below.

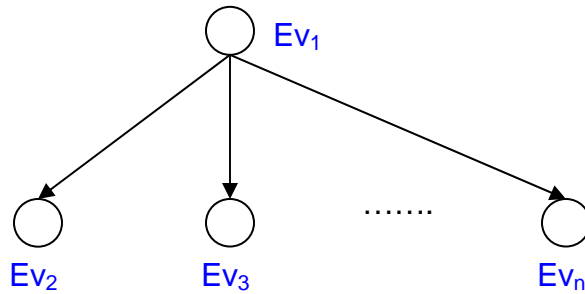


Figure 5. An Event Set.

- **Event Chain** - A sequence of events or event chain is expressed as an event that is composed of a sequence of related events that is temporally ordered as shown in Figure 6 below.

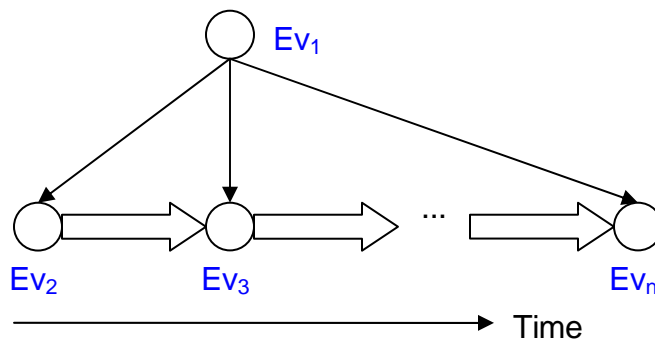


Figure 6. An Event Chain.

- **Alternative Paths** - Separate alternative sets or sequences of events can follow an event, each with some probability of occurrence as shown in Figure 7 below.

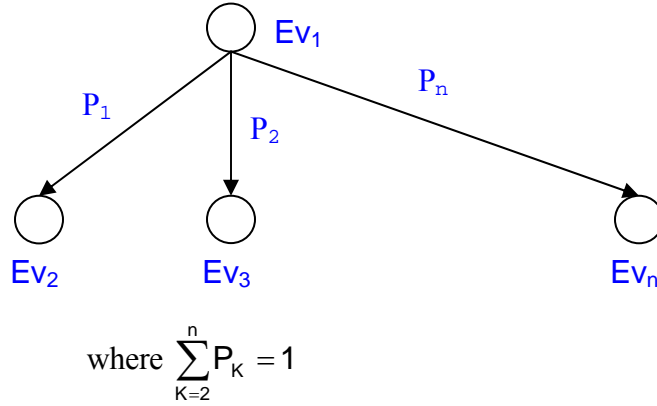


Figure 7. Alternative Events.

In terms of notation:

Notation	Description
Ev_1	An Event
$Ev_1 :: \{ Ev_2, Ev_3 \dots Ev_n \}$	A set of concurrent n events
$Ev_1 :: Ev_2=>Ev_3=>\dots Ev_n$	A sequence of events
$Ev_1 :: \{ Ev_2, Ev_3 \dots Ev_n \} \mid \{ Ev_{n+1}, Ev_{n+2} \dots Ev_m \} \mid \dots$	Separate alternative sets of events
$Ev_1 :: Ev_2=>Ev_3=>\dots Ev_n \mid Ev_{n+1}>Ev_{n+2}>\dots Ev_m \mid \dots$	Separate alternative sequences of events
$Ev_1 :: \{ Ev_2, Ev_3 \dots Ev_n \} \mid Ev_{n+1}>Ev_{n+2}>\dots Ev_m$	Separate alternatives consequence of either a set or sequence of events

Table 1. Relationships Between Events.

2. Concurrent Events

For events that occur concurrently, they can be modeled as a set of events where special synchronizing sub-events common to these events can be defined for information exchange. For example, if sequences $Ev_2=>Ev_3=>Ev_4$ and $Ev_5=>Ev_6=>Ev_7$ are concurrent sequences belonging to the overall system event Ev_1 , in terms of notation:

$$Ev_1 = Ev_2=>Ev_3=>Ev_4 \mid Ev_5=>Ev_6=>Ev_7 \quad (2.11)$$

Then special synchronizing event Ev_s can be defined for both threads, implying:

$$Ev_1 = Ev_2=>Ev_3=>Ev_s=>Ev_4 \mid Ev_5=>Ev_s=>Ev_6=>Ev_7 \quad (2.12)$$

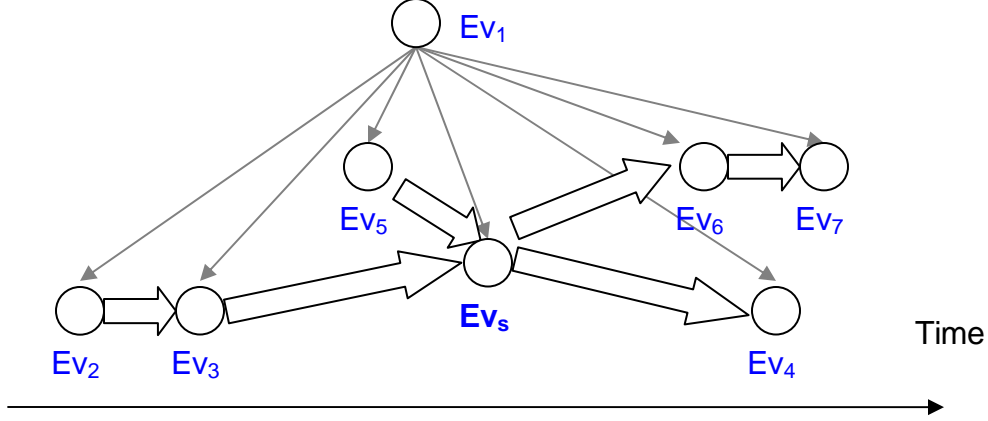


Figure 8. Concurrent Events.

E. STEP 4 - METALS SYNTAX AND SEMANTICS

1. Overview of METALS

METALS is a high level language used to model and built simulation programs in a target language. Specifically, the language models the behavior of simulation programs as a series of events known as event traces. Events can be sequentially ordered and any number of child events can be included within a parent event based on the two fundamental relationships of precedence and inclusion described earlier.

Events are treated as objects (event types) with attributes, and computation over event traces is possible through the manipulation of these attributes. For example, if a simulation program S comprises a sequence of 3 events A , B and C , each with an attribute defined as *Duration*, then the total duration for the simulation program can be obtained by summing up the individual durations of each event as shown by the simple C++ expression below:

```

BEGIN
    S.Duration = A.Duration + B.Duration + C.Duration;
END

```

(2.13)

In equation 2.13, BEGIN and END are METALS key words used to mark the direct insertion of code written in the target language (C++ in this case).

METALS is designed specifically to describe events. METALS leaves the mechanisms for data and attribute manipulation by design largely to the target language.

2. The METALS Event Grammar

The model of the simulation program is formally defined using an event grammar. The event grammar is a set of axioms that describe possible patterns of basic relationships between events of different type during the execution of the simulation program [AUG98].

In METALS, a simulation program comprises the following series of design domain based events:

- Define Name - A name for the simulation is defined along with its description.
- Include Headers -Header files containing global libraries of parameters and routines are included.
- Define and Instantiate Simulation Environments Objects - Classes of objects representing the simulation environment within which entities reside and interact are defined and instantiated.
- Entities - Classes of objects representing real world entities are defined and instantiated.

- Events - Classes of objects representing real world events are defined.
- Chains - Chains of events and their associated patterns of behavior reflecting the real world are defined by the user. At this point, event traces in the problem domain are generation by the overall simulation program.

Expressed in context free grammar, if the preceding design domain chain of events is shown in BNF form¹ below:

```
<Simulation_Program> ::=
    <Title>
    [ <Header>* ]
    [ <World>* ]
    [ <Entity>* ]
    [ <Event>* ]
    [ <Chain>* ]
```

(Rule 1)

Rule 1 is the quintessential formal definition of a simulation program in METALS. Rule 1 mandates that all simulation program must be first named and the process of building the simulation program should then be followed by the occurrences of five different types of events in a prescribed order. This order is influenced by the order in which the simulation program is constructed in the target language and reflects the semantics of this target language (C++). The subsequent event grammar rules for METALS further describes each of the six main event types defined in Rule 1. They are described in detail as follows:

¹ In BNF notation, the meta-symbols "::<=" means "is defined as", "|" means "or", "*" and means "repeated zero or more times". The angle brackets "< >" represents non-terminal symbols that are used to name syntax rules. The square brackets "[]" represents "optional".

2. Rules 2 And 3

`<Title> ::=`
`'SIMULATE'`
`<Simulation_Name>`
`['{' <Description> '}]` (Rule 2A)

`<Simulation_Name> ::= Identifier` (Rule 2B)

`<Description> ::= String` (Rule 2C)

Rule 2 stipulates that a simulation program will begin with the keyword SIMULATE followed by a name defined by the user and optional description enclosed by curly braces.

`<Header> ::= 'INCLUDE' <File_Name>` (Rule 3A)

`<File_Name> ::= Identifier` (Rule 3B)

Rule 3 stipulates that headers files written in the target language can be included into the simulation program by preceding each header file name with the keyword INCLUDE. A code example for Rules 2 and 3 is shown in Figure 9 below:

```
SIMULATE My_Program { Version 1.0 }  
  
INCLUDE My_Filename1  
INCLUDE My_Filename2
```

Figure 9. Code example for Rules 2 and 3.

3. Rule 4 - The WORLD Event

```
<World> ::=  
'WORLD'  
<World_Name>  
'{' <World_Attributes> '}'
```

 (Rule 4A)

```
<World_Name> ::= Identifier
```

 (Rule 4B)

```
<World_Attributes> ::=  
'ROWS' Number  
'COLS' Number  
<Terrain>  
[ <Attributes> ]
```

 (Rule 4C)

```
<Terrain> ::= 'TERRAIN' '{' Identifier* '}'
```

 (Rule 4D)

```
<Attributes> ::= 'ATTRIBS' '{' <CPP_Code> '}'
```

 (Rule A)

```
<CPP_Code> ::= 'BEGIN' CPP_Expression 'END'
```

 (Rule C)

Rule 4 allows the user to define a two dimensional array of objects representing the physical environment within which entities reside and interact. Each of these objects of type <World_Name> has a pre-defined attribute called <Terrain> which allows the users to specify an unlimited number of terrain types. These terrain types are translated to an enumerated data type in the target language (C++). Additional attributes for the objects can be specified directly in the target language (C++) as an option. A code example for Rule 4 is shown in Figure 10 below:

```
WORLD Battlefield  
{  
    ROWS 100  
    COLS 100  
    TERRAIN { Forest Swamp Farm Barren Lake }  
    ATTRIBS {  
        BEGIN  
            double Size;  
        END  
    }  
}
```

Figure 10. Code example for Rule 4.

4. Rule 5 - The ENTITIY Event

```
<Entity> ::=  
'ENTITY'  
<Entity_Name>  
'{' <Attributes> '}' (Rule 5A)  
<Entity_Name> ::= Identifier (Rule 5B)  
  
<Attributes> ::= 'ATTRIBS' '{' <CPP_Code> '}' (Rule A)  
  
<CPP_Code> ::= 'BEGIN' CPP_Expression 'END' (Rule C)
```

Rule 5 allows the user to define objects representing real world entities for the simulation. Attributes for these objects are specified in the target language (C++) directly. A code example for Rule 5 is shown in Figure 11 below:

```
ENTITY Tank  
{  
  ATTRIBS  
  {  
    BEGIN  
      double speed;  
      int status;  
      long ammo_level;  
    END  
  }  
}
```

Figure 11. Code example for Rule 5.

5. Rule 6 - The EVENT Event

```
<Event> ::=
'EVENT'
<Event_Name>
'{' <Event_Attributes>* '}' (Rule 6A)

<Event_Attributes> ::=
['DET' 'DUR' '=' <Duration>] |
['STO' 'MEAN' '=' <Mean>]
[<Attributes>] (Rule 6C)

<Attributes> ::= 'ATTRIBS' '{' <CPP_Code> '}' (Rule A)

<CPP_Code> ::= 'BEGIN' CPP_Expression 'END' (Rule C)

<Event_Name> ::= Identifier (Rule D)
```

Rule 6 is the key language construct for METALS. Rule 6 stipulates how users can define all real world events (design domain events) that are to be simulated in the program. METALS supports 3 standard attributes for an event, namely:

- Stochastic vs Deterministic - An event can be defined as stochastic or deterministic using the keywords 'STO' and 'DET' respectively. This attribute can be used to determine how the duration (time to next event) for is generated for each instantiation of the event object from the event class. For example, a Poisson Random Variate generator routine can be placed inside the constructor of this event class in the target language (C++) to set this duration if the event is defined as stochastic by the user.
- Duration - User can define the duration of the event using the keyword 'DUR'. This value is only valid if the event is defined as deterministic by the user.
- Mean - User can define the mean duration of the event using the keyword 'MEAN' that can be used in random variate generation routines. This value is only used if the event is defined as stochastic by the user.

Additional attributes for the objects can be specified directly in the target language (C++) as an option. A code example for Rule 6 is shown in Figure 12 below:

```
EVENT Tank_Move
{
    STO MEAN = 8
    ATTRIBS
    {
        BEGIN
            int startrow;
            int startcol;
            int endrow;
            int endcol;
            double fuel_consumed;
        END
    }
}
```

Figure 12. Code example for Rule 6.

In the above example, the `Tank_Move` event is defined with an attributed `fuel_consumed`. This attribute is used to track the resource consumption level in the simulation, the resource in this case being `fuel`. Although a total of 5 different attributes has been defined in the code, the event class when translated into the target language (C++) will contain additional 'hidden' standard attributes, namely the event's name, event's type (stochastic or deterministic), event's mean duration and event's duration. Specific details of this translation will be described when the code generator for METALS is presented in the next chapter.

6. Rule 7 - The CHAIN Event

```
<Chain> ::=  
'CHAIN'  
<Chain_Name>  
'{ ' <Rule>* '}'
```

 (Rule 7A)

```
<Chain_Name> ::= Identifier
```

 (Rule 7B)

```
<Rule> ::= <Event_Name> [ ':' <Pattern>* ] ';' 
```

 (Rule 7C)

```
<Event_Name> ::= Identifier
```

 (Rule D)

All the preceding set of rules that has been described so far has enabled the various simulation environments, entities and events to be customary defined. The actual behavior of the simulation program, specifically in terms of how the events unfold in the simulation and how the entities will interact during has yet to be determined.

Rule 7 provides the necessary language constructs that will enable the logical flow of events and their mutual interactions under specific conditions to be defined. Rule 7 centers around the notion of event chains and event rules.

An event chain represents quite literally a chain or series of events. In METALS, each event chain is represented by a master event START which encompasses all other possible events within the chain. Figure 13 below illustrates this.

```
Start_Rule : Event_A Event_B Event_C;  
  
Event_A_Rule : Event_D Some_E_Action;  
  
Event_B_Rule : Some_B_Action;  
  
Event_C_Rule : Some_C_Action;  
  
Event D_Rule : Some_D_Action;
```

Figure 13. Code example for Rule 7.

The behavior of each event within the chain is defined as an event rule. Every event type defined by the user previously must have a corresponding event rule that defines its behavior. The special START event rule associated with the master START event type therefore is the entry point of execution for each event chain defined in the simulation program.

In Figure 14, The behavior of the master event START is defined in the START_RULE which stipulates that the entire simulation comprises 3 events executed in sequence, Event A followed by Event B followed by Event C. Subsequent rules are then defined for each event type that has been previously defined. For example, the rule for Event A stipulates that it comprises 2 events, Events D followed some actions, and specific rules for Events B, C and D describes in target language exactly what happened during their occurrences.

The previously described two binary rules of precedence and inclusion which governs the relationship between events are fully supported by METALS. In other words, the event chain given in Figure 13 can be re-written in its equivalent forms presented in Figures 14 and 15.

```
Start_Rule : Event_D Some_E_Action Event_B Event_C;  
Event_B_Rule : Some_B_Action;  
Event C_Rule : Some_C_Action;  
Event_D_Rule : Some_D_Action;
```

Figure 14. An Event Chain - Part 2.

```
Start Rule : Some_D_Action Some_E_Action  
Some_B_Action Some_C_Action;
```

Figure 15. An Event Chain - Part 3.

In order to improve the intuitiveness and usability of the language, the syntax

presented in Rule 7 stipulates that rule name on the left hand side of each rule expression should be exactly the event name. In addition, the METALS compiler will always assume that the very first rule declared by the user within an event chain is the START event and be designated as the entry point for the simulation automatically. Therefore, there is no keyword START or START_RULE mandated by the syntax for METALS.

Another important aspect in Rule 7 is the language constructs that describes the standard patterns of behavior for events that the user can use in writing each rule. Subsequent rules that support these standards patterns² include rules for Iteration, Loop, Conditional, Alternatives and Action. There is no special construct supporting the concurrency of event is designed at this time. It is felt that the event chain construct which allows two sets of independent, non-synchronizing and concurrent events to be defined and executed is sufficient for the first version of the language.

A code example for Rule 7 is shown in Figure 16 below:

```
START_SIM : Deploy Search Engage

Deploy : BEGIN
    Tank.StartRow = 0;
    Tank.StartCol = 0;
END;

Search : WHILE ( Tank.found_enemy == false )
    { Tank_Move };

Tank_Move : BEGIN
    Tank.CurrentRow += 1;
    Tank.CurrentCol += 1;
    if (Tank.Contact == Enemy_Tank)
        Tank.found_enemy == true
    END;

Engage : BEGIN Tank.Ammo_Level -= 1; END;
```

Figure 16. An Event Chain - Part 3.

² A detailed description of these patterns has already been presented in Chapter 1 at page 11.

All rule statements must end with the semi-colon character ';'.

7. Rule 8 - The PATTERN Event

```
<Pattern> ::=  
<Simple> |  
<Action> |  
<Conditional> |  
<Alternative> |  
<Loop> |  
<Iteration> |  
<Group> (Rule 8A)
```

Rule 8A defines 7 standard patterns that can be used to describe the behavior of events in a simulation.

a. The SIMPLE Pattern

```
<Simple> ::= <Event_Name> (Rule 8B)
```

```
<Event_Name> ::= Identifier (Rule D)
```

The SIMPLE Pattern is used to substitute code describing the behavior of an event on the right hand side of an event rule with the name of the event itself. In the code example shown in Figure 17, both Event_B and Event_C are simple patterns. In the example, the specific implementation for both Event_B and Event_C can be defined separately without affecting the implementation for Event_A_Rule.

```
Event_A_Rule : Event_B Event_C;
```

Figure 17. The SIMPLE Pattern.

b. *The ACTION*

`<Action> ::= <CPP_Code>` (Rule 8C)

`<CPP_Code> ::= 'BEGIN' CPP_Code 'END'` (Rule C)

The ACTION Pattern can be used to insert code written in the target language anywhere within an event rule. The specific code must be placed between the keywords 'BEGIN' and 'END'. An application of the ACTION pattern is shown in the code example in Figure 18 below:

```
Event_A_Rule :  
Event A  
Event_B  
BEGIN cout << "Simulation Commenced"; END  
Event_C;
```

Figure 18. The ACTION Pattern.

c. *The CONDITIONAL Pattern*

`<Conditional> ::=`
`'WHEN' '(' Boolean_Expression ')' '{' <Pattern>* '}'`
`['ELSE' '{' <Pattern>* '}']` (Rule 8D)

The CONDITIONAL Pattern provides the language construct that allows the simulation program to make a decision on whether to proceed with some pre-defined events based on some pre-defined condition. Conditional METALS statements must begin with the keyword 'WHEN' followed by the a Boolean expression (written in the target language) enclosed in brackets. The pre-defined events to be executed when the Boolean expression returns true should be enclosed in curly brackets. An optional alternative set of pre-defined events to be executed on the falsity of the Boolean expression can be specified by the keyword 'ELSE'. An application of the ACTION pattern is shown in the code example in Figure 19 below:

```
Event_A_Rule :
WHEN ( Event_B.Duration > 2 ) { Event_C }
ELSE { Event_D };
```

Figure 19. The CONDITIONAL Pattern.

d. The ALTERNATIVE Pattern

```
<Alternative> ::=
'DECIDE' '(' <Outcome>* '|' ' ' ')' (Rule 8E)
```

```
<Outcome> ::=
['P' '(' <Probability> ')' ] <Pattern>* (Rule 8F)
```

```
<Probability> := Number between 0 and 1 (Rule 8G)
```

The ALTERNATIVE Pattern provides the language construct that allows the simulation program to choose to proceed with some pre-defined events amongst a set of alternatives or outcomes. Alternative METALS statements must begin with the keyword 'DECIDE' followed by a series of possible outcomes separated by the symbol '|'.

The probability of occurrence for each outcome can be optionally specified with the prefix 'P' followed by a value between 0 and 1 enclosed in braces. If no probability is specified in this way, the METALS will assign equal probabilities to all outcomes.

A code example for the ALTERNATIVE Pattern where all outcomes have equal probabilities of occurrence is shown in Figure 20 below.

```
Event_A_Rule :
DECIDE ( Event_B | Event_C | Event_D );
```

Figure 20. The ALTERNATIVE Pattern - Part 1.

Another code example for the ALTERNATIVE Pattern where specific probabilities of occurrence are specified is shown in Figure 21 below:

```
Event_A_Rule :  
DECIDE ( P(0.5) Event_B |  
         P(0.3) Event_C |  
         P(0.2) Event_D );
```

Figure 21. The ALTERNATIVE Pattern - Part 2.

The number zero character can be omitted when specifying probability values. The equivalent code for the example shown in Figure 21 is shown in Figure 22 below:

```
Event_A_Rule :  
DECIDE ( P(.5) Event_B |  
         P(.3) Event_C |  
         P(.2) Event_D );
```

Figure 22. The ALTERNATIVE Pattern - Part 3.

The sum of all probabilities must equate to the value 1. If the sum of all probabilities exceed 1, then only events up to point before the sum of all probabilities exceed 1 will be recognized. In the code example shown in Figure 23, the sum of the probabilities exceed one at the outcome Event_D. METALS will therefore assign a probability of 0.2 to Event_D and ignore Event_E altogether. The equivalent code for the example shown in Figure 23 is shown in Figure 24 below.

```

Event_A_Rule :
DECIDE ( P(.5) Event_B |
        P(.3) Event_C |
        P(.3) Event_D |
        P(.2) Event_E );

```

Figure 23. Special case for ALTERNATIVE Pattern - Part 1A.

```

Event_A_Rule :
DECIDE ( P(.5) Event_B |
        P(.3) Event_C |
        P(.2) Event_D );

```

Figure 24. Special case for ALTERNATIVE Pattern - Part 1B.

If the sum of all probabilities is less than 1, then the last outcome will be assigned a probability that will set this sum to unity. In the code example shown in Figure 25, the sum of the probabilities is short of one. METALS will therefore assign a probability of 0.5 to Event_D. The equivalent code for the example shown in Figure 25 is shown in Figure 26 below:

```

Event_A_Rule :
DECIDE ( P(.3) Event_B |
        P(.2) Event_C |
        P(.1) Event_D );

```

Figure 25. Special case for ALTERNATIVE Pattern - Part 2A.

```

Event_A_Rule :
DECIDE ( P(.3) Event_B |
        P(.2) Event_C |
        P(.5) Event_D );

```

Figure 26. Special case for ALTERNATIVE Pattern - Part 2B.

e. The LOOP Pattern

```
<Loop> ::=  
'WHILE' '(' Boolean_Expression ')'  
'{' <Pattern>* '}'
```

 (Rule 8H)

The LOOP Pattern is a repetition language construct that allows the simulation program to repeat a set of events while some pre-defined condition remains true. Loop METALS statements must begin with the keyword 'WHILE' followed by the a Boolean expression (written in the target language) enclosed in brackets. The pre-defined events to be executed when the Boolean expression returns true should be enclosed in curly brackets. An application of the LOOP pattern is shown in the code example in Figure 27 below:

```
<  
Event_A_Rule :  
WHILE (Simulation.Terminate == false)  
{ Event_B Event_C };
```

Figure 27. The ALTERNATIVE Pattern.

f. The ITERATION Pattern

```
<Iteration> ::=  
'REPEAT' '(' <Operator> Expression ')'  
'{' <Pattern>* '}'
```

 (Rule 8I)

```
<Operator> ::= ( '<' | '<=' | '=' )
```

 (Rule B)

The ITERATION Pattern is another repetition language construct that allows the simulation program to repeat a set of events for a specified number of times. Unlike the LOOP Pattern where the Boolean expression is specified in the target language, the ITERATION Pattern enable the exact number of repetitions executed to be stochastic or deterministic by prefixing the number of repetitions (that can be computed from an expression written in the target language) with one of three pre-defined operators.

In the code example shown in Figure 28 below, the operator '=' is specified. This means that Event_B will be executed exactly 10 times.

```
Event_A_Rule :  
REPEAT (=10) { Event_B };
```

Figure 28. The ITERATION Pattern - Part 1.

In the code example shown in Figure 29 below, the operator '<=' is specified. This means that Event_B will be executed any number of times between 0 to 10.

```
Event_A_Rule :  
REPEAT (<=10) { Event_B };
```

Figure 29. The ITERATION Pattern - Part 2.

In the code example shown in Figure 30 below, the operator '<' is specified. This means that Event_B will be executed any number of times between 0 to 9.

```
Event_A_Rule :  
REPEAT (<10) { Event_B };
```

Figure 30. The ITERATION Pattern - Part 3.

g. The GROUP Pattern

`<Group> ::= '(' <Pattern>* ')'` (Rule 8J)

The GROUP Pattern allows a series of events to be grouped together and enclosed between parentheses. The program will treat this group of events as a contiguous block. The GROUP Pattern can be used to create composite outcomes in ALTERNATIVE statements such as in the example shown in Figure 31 below. An equivalent code is shown in Figure 32.

```
Event_A_Rule :  
DECIDE ( P(.5) Event_B |  
        P(.3) ( Event_C Event_D Event_E ) |  
        P(.2) Event_D );
```

Figure 31. The GROUP Pattern.

```
Event_A_Rule :  
DECIDE ( P(.5) Event_B |  
        P(.3) Composite_Event |  
        P(.2) Event_D );  
  
Composite_Event_Rule : Event_C Event_D Event_E;
```

Figure 32. Composite Events.

THIS PAGE INTENTIONALLY LEFT BLANK

III. IMPLEMENTATION OF METALS

A. AN OVERVIEW OF THE METALS COMPILER

The METALS compiler comprises three main components that operate in sequence to automatically transform code written in the high level METALS language described in depth in the preceding chapter to a simulation program generated in the target language. The three components are the Lexical Analyzer, the Parser and the Code Generator.

A Lexical Analyzer will first read the high level source code from a file as a stream of characters and strings, and then produces a stream of symbols called "lexical tokens" that can be handled by the Parser more efficiently.

A Parser will next analyze the grammatical structure of the token stream with respect to the formal context free grammar for METALS. The METALS parser employs the top-down recursive descent parsing technique. In this top-down process, the parser will attempt to check the syntax of the token stream from left to right, reading each token from the input token stream and then engages in pattern matching to match the token with the terminals from the pre-defined grammar. Each grammar rule corresponds to a recursive function that is used by the parser to construct the parse tree. The resulting parse tree or code in intermediate form is then fed to the Code Generator.

A Code generator will first synthesize the parse tree by assigning useful values (annotate) to every node on the tree in such a way that either the value at one node determines the values for its children or vice versa. These values can subsequently be used by specific routines to output equivalent code in the target language.

The overall process of compiling a METALS program is illustrated in Figure 33 below:

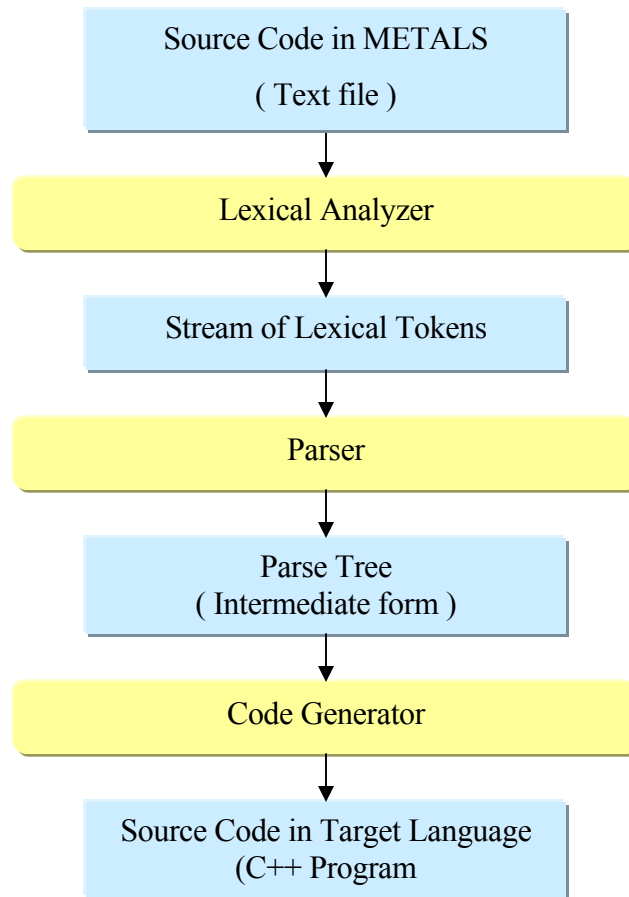


Figure 33. Components of the METALS Compiler.

B. COMPILER DEVELOPMENT TOOL

1. The RIGAL Compiler Development Language And Toolkit

The METALS compiler is developed in the RIGAL compiler construction language [RIG1][RIG2] that is first developed at the University of Latvia, Institute of Mathematics and Computer Science in 1987. The main data structures in the language are atoms, lists and labeled trees. Control structures in RIGAL are based on advanced pattern matching. The METALS parser and code generator was programmed in this language in short and readable form using the RIGAL development toolkit [RIG3]. The built in RIGAL

lexical analyzer from the toolkit is used as the first component to the compiler.

The text of a RIGAL program is a sequence of tokens - variables, atoms, lists, trees, keywords, special symbols and rules. Tokens may be surrounded by any number of blanks. A comment is any string of symbols that begins with the symbol `--`. The end of the comment is the end of the line. An example of a simple RIGAL program is shown in Figure 34 below:

```
#Sum -- rule for addition of two numbers
    $N1 -- the first number
    $N2 -- the second number
    / RETURN $N1 + $N2 / -- return of the result
##
```

Figure 34. A Simple RIGAL Program.

2. RIGAL Variables

Variables in RIGAL are prefixed with the symbol `$` followed by an identifier. A value can be assigned to a variable with the `:=` assignment operator. In RIGAL, variables have no data types and the value of a variable can either be an atom, a list or a tree. For example, the assignment statement `$Unit := Soldier` means that the atom `Soldier` is now the value of the variable `$Unit`.

3. RIGAL Atoms, Lists And Trees

The RIGAL Language contains only three types of data structures, namely Atoms, Lists and Trees.

a. Atoms

The most elementary data structure is the ATOM, which is used to represent a text string (identifier) or an integer (number). An atom can be compared with another atom to see if both of them are identical. This comparison forms the basis of the pattern matching functionality that is essential in the programming of a parser or code generator.

Identifier atoms can be written directly as in Apple, Orange, Banana or in quotes as in 'Apple', 'Orange', '123'. Numerical atoms are written directly as 123. It is worth noting that the identifier '123' is not equivalent to the numerical value 123. Besides identifier and numerical atoms, two other special types of atoms also exist in RIGAL, namely the NULL atom and the T atom. The NULL atom which is yielded by an error, or used to represent an error, an empty list, an empty tree and the Boolean value 'FALSE'. The T atom is yielded by logical operations and represents the value 'TRUE'.

b. Lists

A list is an ordered sequence of atoms, trees or other lists. In RIGAL, a list can be created using a special List constructor denoted by a pair of (. and .). special symbols. For example, the statement (. A B C .) creates a list comprising three atoms A, B and C. Figure 35 below illustrates a list in graphical form.

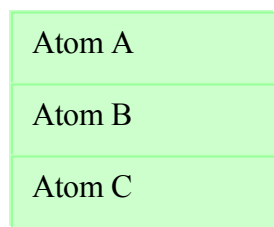


Figure 35. A Simple List Comprising 3 Atoms.

Individual elements of a list can be referenced by indexing. If the variable \$List is a list comprising 3 atoms A B and C, then the expression \$List[1] represents Atom A. For indices that are out of range, in other words greater than the total number of elements in the list, the atom NULL is automatically assigned as value to the expression. For example, using the same example, the expression \$List[10] will mean that the

NULL atom will be its value. A negative index -N will represent the Nth element counting from the end of the list. For example, the expression `$List[-1]` will refer to Atom C.

An additional element can be inserted into the list using the operator denoted by the `!.` special symbol. For example the statement `(. A B .) !. C` will yield the list `(. A B C .)`.

Two lists can be concatenated to form a new list using the `!!` special symbol. For example, the statement `(. A B .) !! (. C D .)` will yield the list `(. A B C D .)`.

c. *Trees*

A tree is a recursive nodal data structure that holds data in nodes that are hierarchically connected. A tree begins with the parent node that contains zero or more child nodes or leaves each connected by a single arch or link. In RIGAL, a tree can be created using a special Tree constructor by a pair of `<.` and `.>` tree symbols and `:` tree selector symbol. For example, the statement `<. A : B, C : D .>` will create a tree with two leaves, each terminating at the atoms B and D respectively via selectors or links A and C. An equivalent visual representation is shown in the diagram below:

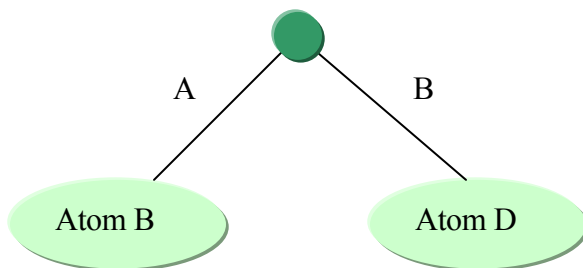


Figure 36. A Simple Tree Comprising 2 Atoms.

Elements before the `:` symbol in the tree constructor are named selectors. In the example shown above, A and B are selectors for Atom B and Atom D respectively. The

tree constructor is computed from left to right.

Multi layer trees can be built using the tree constructor. For example, the statement `<. A : B, C : <. D : E, F : G .> .>` will create a tree with the following visual representation:

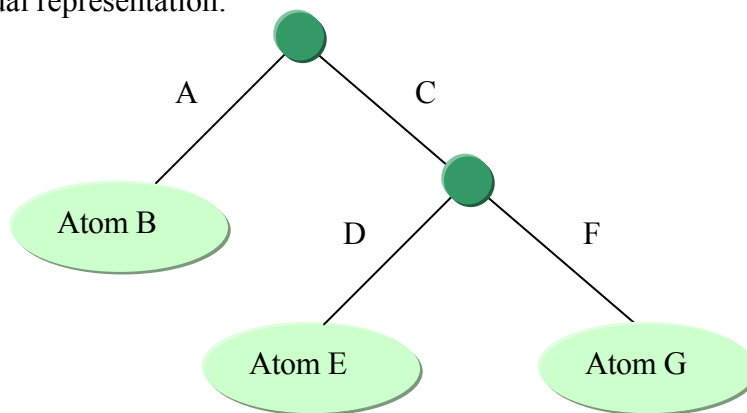


Figure 37. A Multi-layer Tree Comprising 3 Atoms.

The leaves of a tree can contain a list of atoms. For example, the statement `< A : B, C : <. D : E, F : (. G H I .) .> .>` will create a tree with the following visual representation.

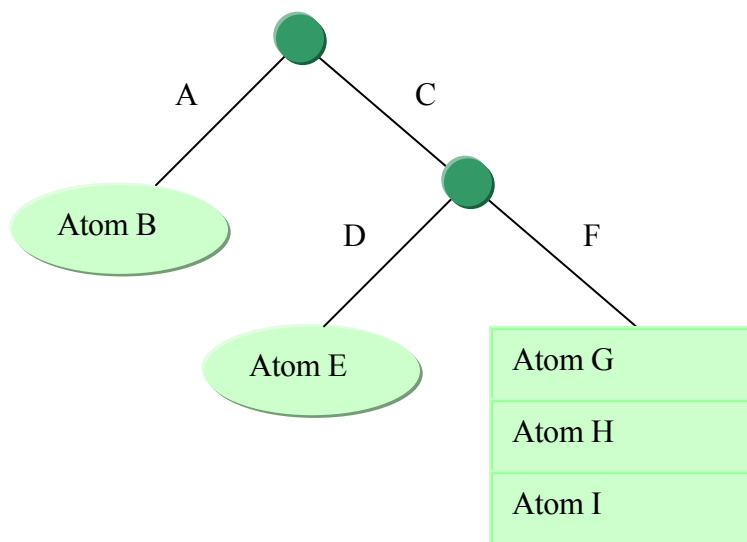


Figure 38. A Multi-layer Tree Comprising 5 Atoms.

An additional leaf or tree branch can be inserted into a tree using the operator denoted by the **++** special symbol. In other words, if T1 and T2 are trees with some branches, the statement **T1 ++ T2** means that all tree branches of T2 will be added to the tree T1 one by one. For example, the statement **<. A : B .> ++ <. C : D, E : F .>** will yield the tree **<. A : B, C : D, E : F .>** as illustrated visually below:

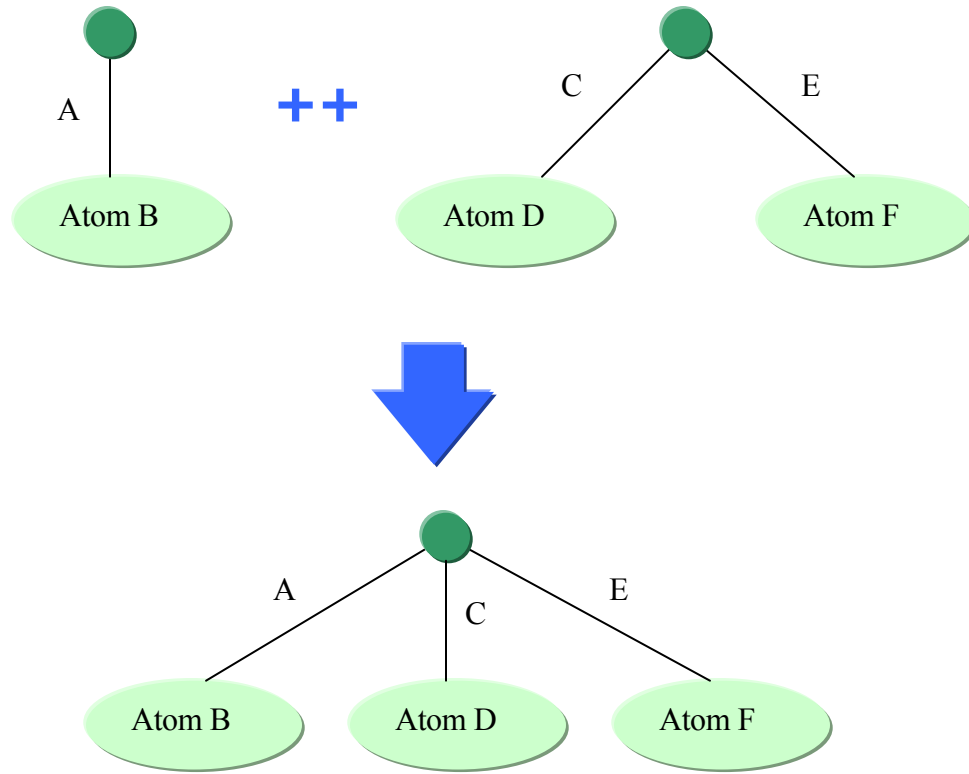


Figure 39. Tree Addition.

For any tree addition **T1 ++ T2**, if in the tree T1 there is already a branch with the same selector of another branch in T2, the T1 branch is substituted by the new T2 branch. Therefore, the operation **++** is not commutative.

3. RIGAL Pattern Matching Rules

The RIGAL Language uses Rules to check whether an input token or a series of tokens (source code) complies with some pre-defined grammar. For this purpose, each rule is constructed with a certain pattern that reflects the underlying grammar. Input tokens that are handled by a rule are matched with this pattern. A RIGAL rule can also perform computations and return values or objects (atoms, lists, trees). A rule call will end with either a success (where pattern matching is successful) or failure. A rule call can be called recursively or be called inside another rule.

A rule definition first begins with its name prefixed by the special symbol `#`. A pattern of objects follows before the special symbols `##` marks its end. An example of a simple rule is given below:

```
#Rule_A
  A
  ( . B C . )
  < . D : E , F : G . >
  / RETURN ( . A B C E G . ) /
##
```

Figure 40. A Simple RIGAL Rule.

In the simple RIGAL rule shown in Figure 40, the rule `Rule_A` expects to see an Atom first, followed by a list and finally a tree. As long as the input stream of tokens matches the pattern prescribed in the rule, pattern matching will continue until the end of the rule, else the rule will failure the same input stream of tokens will be compared with the next rule. Computation statements are enclosed using a pair of `/` symbols, which in the example above, returns a list of all atoms from the input stream.

Therefore, if the input tokenized source code contains the tokens shown in Figure 41 below, than the pattern matching is successful and the rule will return the list of atoms `(. Ship Submarine_A Submarine_B Destroyers Minehunters .)`.

```
Ship (. Submarine_A Submarine_B .)
<. Task_Force_A : Destroyers,
   Task_Force_B : Minehunters .>
```

Figure 41. An example of an input stream of tokens.

However if the input tokenized source code is as indicated in Figure 42, then the rule will fail because the second token is a tree rather than an expected list.

```
Ship <. Task_Force_A : Destroyers,
      Task_Force_B : Minehunters .>
(. Submarine_A Submarine_B .)
```

Figure 42. An example of code with syntax error.

Rule definitions form the basis of the development of the parser and code generator for METALS.

C. THE METALS LEXICAL ANALYZER

For the current implementation of METALS, the lexical analyzer function in the RIGAL toolkit is used for tokenizing the input source code. The RIGAL lexical analyzer is invoked programmatically using the special built-in RIGAL rule #CALL_PAS.

In the example given in below, the input source code in Figure 43 is parsed into the rule #CALL_PAS. The resulting output stream of tokens is shown in Figure 44.

```

SIMULATE Tokenization_Example { Version 1 }

EVENT Start_Flipping

CHAIN Algorithm
{
    Start_Flipping :
}

```

Figure 43. Input source code.

```

(. 'SIMULATE' 'Tokenization_Example' '{' 'Version' 1
'{' 'EVENT' 'Start_Flipping' 'CHAIN' 'Algorithm' '{'
'Start_Flipping' ':' '}' .)

```

Figure 44. Output list of tokens.

The complete implementation in RIGAL language for the lexical analyzer is given in Appendix I.

D. THE METALS PARSER

1. The Main #Parse Rule

The METALS parser is constructed using a set of recursive RIGAL rules whose patterns are based on the METALS context free grammar. The parser receives an input list of tokens from the RIGAL lexical analyzer and uses the main #Parse Rule to generate an equivalent parsed tree. Parsing is top-down recursive using the LL(1) technique with backtracking. This means that tokens are read from Left to right to produce a Leftmost deviation 1 token at a time. The backtracking feature will be explained in the description for the rule #Pattern. The expected input pattern for a syntactically correct METALS source code is in the form shown in Figure 45 below:

```

( .
  $Title := #Title
  [(* $Headers !.:= #Header *)]
  [(* $Worlds !.:= #World *)]
  [(* $Entities !.:= #Entity *)]
  [(* $Events !.:= #Event *)]
  [(* $Chains !.:= #Chain *)]
. )

```

Figure 45. Expected Input Pattern for a METAL programs.

In the input token stream shown above, the parser expects a title following by 5 lists of user defined headers, worlds, entities, events and event chains in this order. 6 specific rules are used to see if the current token in the stream pattern matches as a title, a header, a world, an entity, an event or a chain. Except for the title token which is immediately stored in an atom **\$Title**, all other successfully matched tokens are then accumulated into their respective lists. In terms of program execution flow, the main **#Parse** rule for the parser calls 6 other rules as illustrated in Figure 46 below:

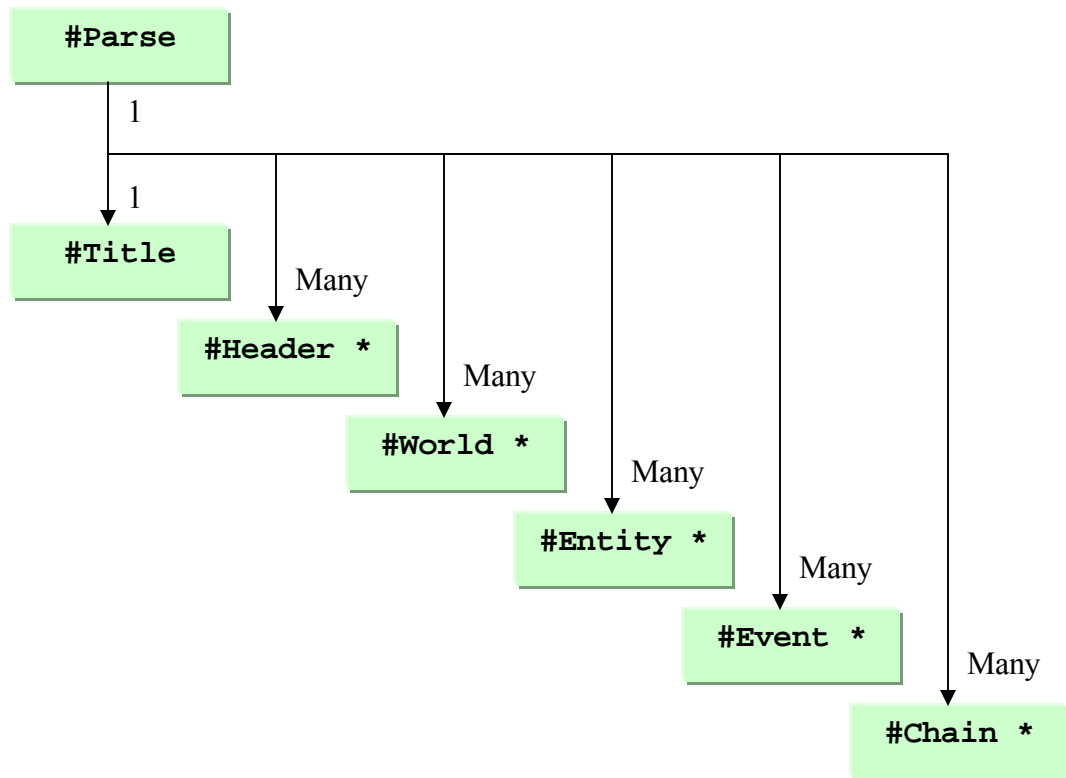


Figure 46. Program Execution Flow for **#Parse** Rule.

The resulting output returned by the **#Parse** rule is in the form of a tree (parsed tree). The structure of the tree is shown in Figure 47.

```
<. Title: $Title,  
  Headers: $Headers,  
  Worlds: $Worlds,  
  Entities: $Entities,  
  Events: $Events,  
  Chains: $Chains .>
```

Figure 47. The METALS Parsed Tree Structure.

The equivalent visual notion for the parsed tree is shown in Figure 48 below:

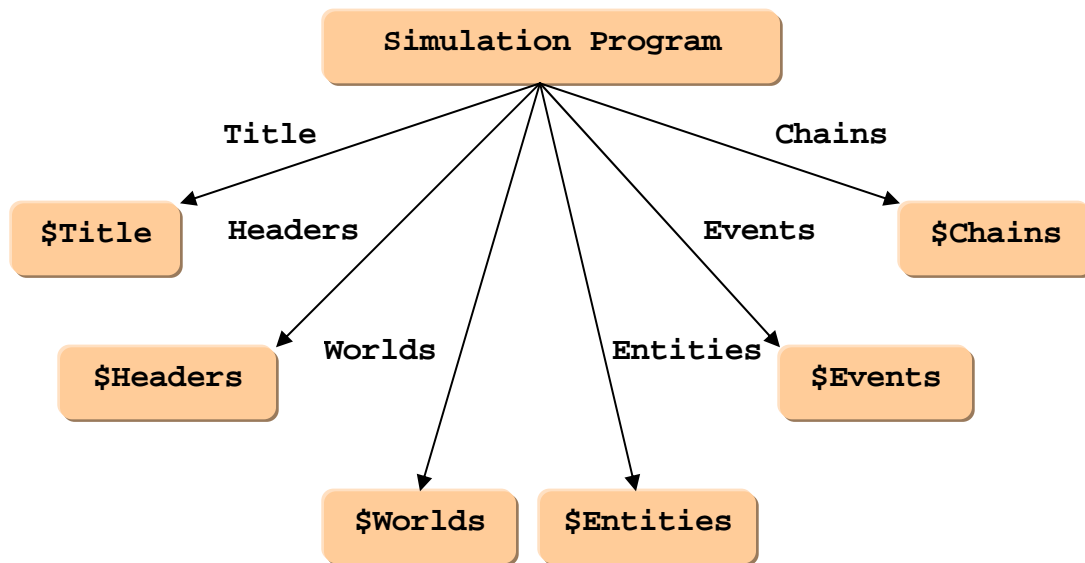


Figure 48. The METALS Parsed Tree.

The **#Parse** rule is derived from METALS grammar Rule 1.

2. The #Title Rule

The title rule expects the input token stream to contain the pattern shown in Figure 49 below. The **#Title** rule is derived from METALS grammar Rule 2.

```
'SIMULATE'  
$Title  
['{' ['$Description := #Plain_Text] '}]'
```

Figure 49. Expected Input Pattern for **#Title**.

The rule expects to detect the use of the keyword **SIMULATE** followed by an identifier and finally some option description in plain text. In terms of program execution flow, the main **#Title** rule for the parser can call the **#Plain_Text** rule to collect a stream of plain text in a list as illustrated in Figure 50 below:

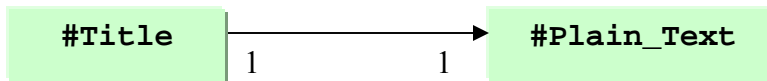


Figure 50. Program Execution Flow for **#Title** Rule.

The resulting output returned by the **#Title** rule is in the form of a tree whose structure is shown in Figure 52. This tree data structure is assigned to the **\$Title** variable in the main **#Parse** rule.

```
<. Title: $Title,  
  Description: $Description .>
```

Figure 51. Output returned by the **#Title** rule.

The equivalent visual notion for the parsed tree is shown in Figure 52 below:

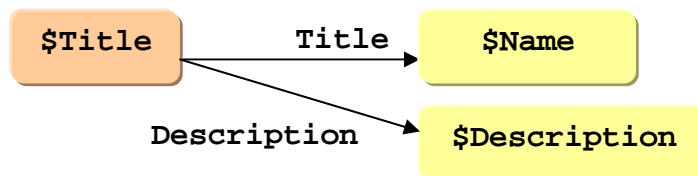


Figure 52. The **#Title** Parsed Tree Structure.

3. The #Header Rule

The header rule expects the input token stream to contain the pattern shown in Figure 53 below. The #Header rule is derived from METALS grammar Rule 3.

```
'INCLUDE' $Header_Name
```

Figure 53. Expected Input Pattern for #Title.

The rule expects to detect the use of the keyword INCLUDE follow by an identifier. The rule does not call any other rules and returns an identifier atom as shown in Figure 54. This identifier is collected in the list \$Headers in the main #Parse rule.

```
$Header_Name
```

Figure 54. Output returned by the #Title rule.

The equivalent visual notion for the #Header rule is shown in Figure 55 below:

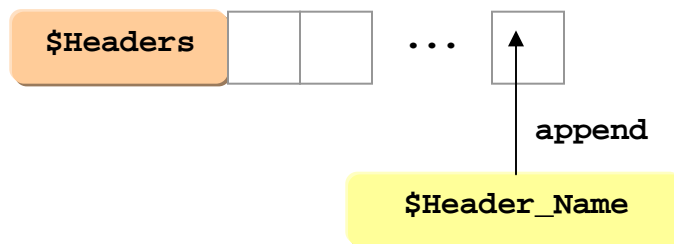


Figure 55. The #Header Structure.

3. The #World Rule

The header rule expects the input token stream to contain the pattern shown in Figure 56 below. The **#World** rule is derived from METALS grammar Rule 4.

```
'WORLD'  
$World_Name  
{  
  'ROWS' '=' $MaxRows  
  'COLS' '=' $MaxCols  
  'TERRAINS' '='  
  '(' (* $Terrains !.:= S'($$<> ')') *) ')'  
  ['ATTRIBS' '=' '(' $Attributes := #CPP_Code ')']  
}
```

Figure 56. Expected Input Pattern for **#World**.

The rule expects to detect the use of the keyword **WORLD** followed by an identifier representing the name of the world object and finally a detailed specification of its attributes using the keywords ROWS, COLS, TERRAINS and ATTRIBS. To obtain the attributes, the rule can call the **#CPP_Code** rule to collect a stream of text representing source code written in the target language in a list as illustrated in Figure 57 below:

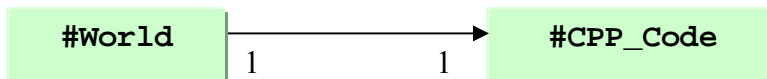


Figure 57. Program Execution Flow for **#World** Rule.

The resulting output returned by the **#World** rule is in the form of a tree whose structure is shown in Figure 58. This tree data structure is inserted into the **\$World** list in the main **#Parse** rule.


```
<. World_Name: $World_Name,
  MaxRows: $MaxRows,
  MaxCols: $MaxCols,
  Terrains: $Terrains,
  Attributes: $Attributes .>
```

Figure 58. Output returned by the `#World` rule.

The equivalent visual notion for the parsed tree is shown in Figure 59 below:

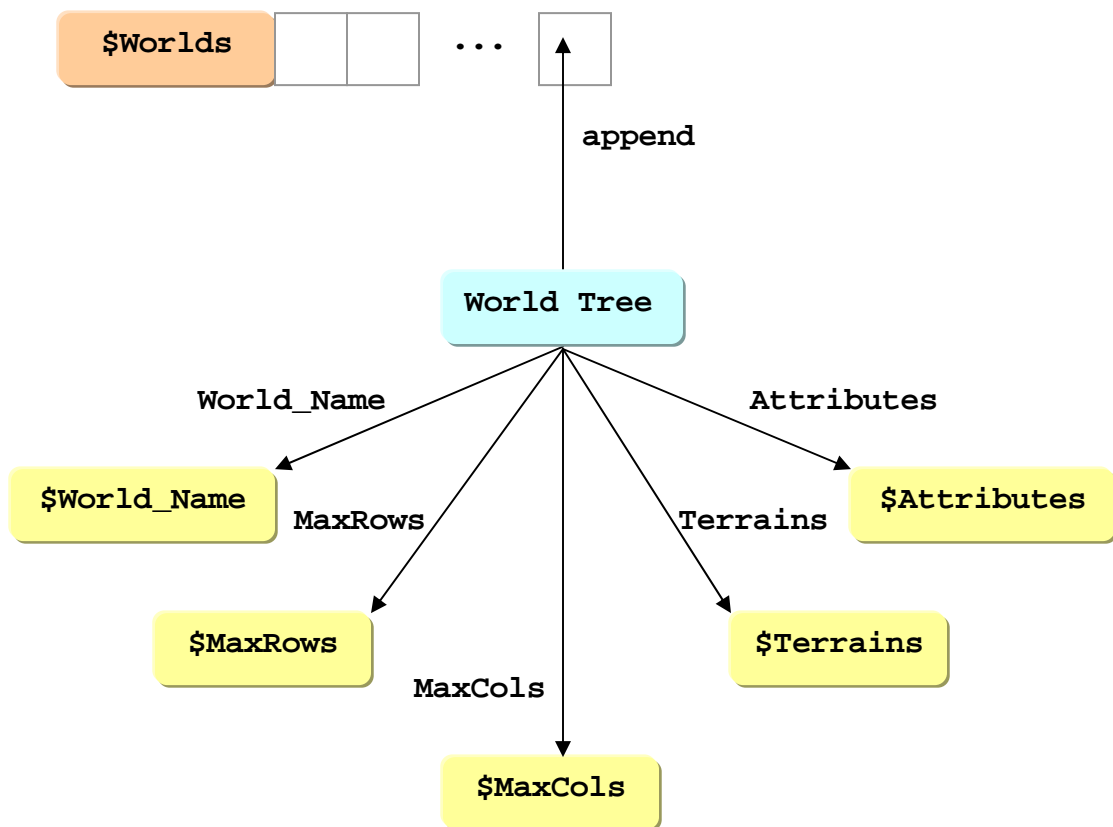


Figure 59. The `#World` Parsed Tree Structure.

4. The #Entity Rule

The entity rule expects the input token stream to contain the pattern shown in Figure 60 below. The #Entity rule is derived from METALS grammar Rule 5.

```
'ENTITY'  
$Entity_Name  
['{' [$Entity_Attributes := #CPP_Code] '}]
```

Figure 60. Expected Input Pattern for #Entity.

The rule expects to detect the use of the keyword ENTITY followed by an identifier representing the name of the entity object and finally a detailed specification of its attributes. To obtain the attributes, the rule can call the #CPP_Code rule to collect a stream of text representing source code written in the target language in a list as illustrated in Figure 61 below:

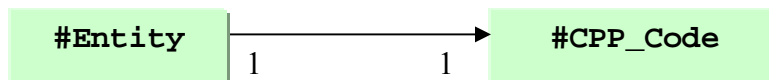


Figure 61. Program Execution Flow for #Entity Rule.

The resulting output returned by the #Entity rule is in the form of a tree whose structure is shown in Figure 62. This tree data structure is inserted into the \$Entities list in the main #Parse rule.

```
<. Entity_Name: $Entity_Name,  
Entity_Attributes: $Entity_Attributes .>
```

Figure 62. Output returned by the #World rule.

The equivalent visual notion for the parsed tree is shown in Figure 63 below:

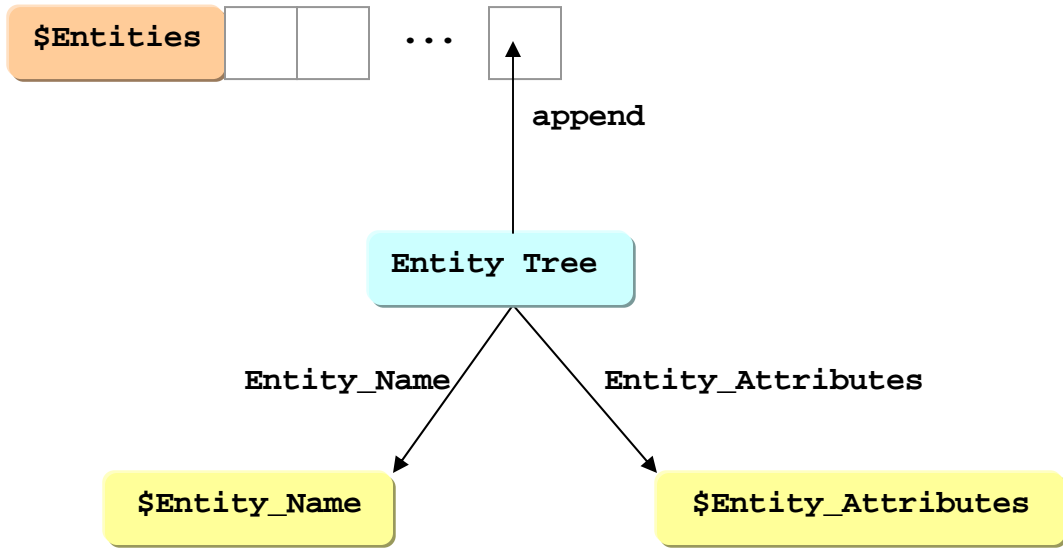


Figure 63. The **#Entity** Parsed Tree Structure.

5. The **#Event** and **#Event_Attributes** Rules

a. The **#Event** Rule

The event rule expects the input token stream to contain the pattern shown in Figure 64 below. The **#Event** rule is derived from METALS grammar Rule 6.

```

'EVENT'
$Event_Name
['{' [$Event_Attributes := #Event_Attributes]
}']

```

Figure 64. Expected Input Pattern for **#Event**.

The rule expects to detect the use of the keyword **EVENT** followed by an identifier representing the name of the event object and finally a detailed specification of its attributes. To obtain the attributes, the rule can call the **#Event_Attributes** rule to collect a stream of text representing source code written in the target language in a list

as illustrated in Figure 65 below:

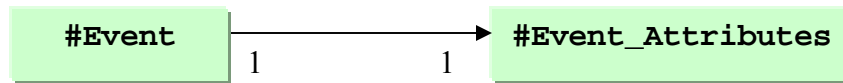


Figure 65. Program Execution Flow for **#Event** Rule.

The resulting output returned by the **#Event** rule is in the form of a tree whose structure is shown in Figure 66. This tree data structure is inserted into the **\$Events** list in the main **#Parse** rule.

```

<. Event_Name: $Event_Name,
   Event_Attributes: $Event_Attributes .>
  
```

Figure 66. Output returned by the **#Event** Rule.

The equivalent visual notion for the parsed tree is shown in Figure 67 below:

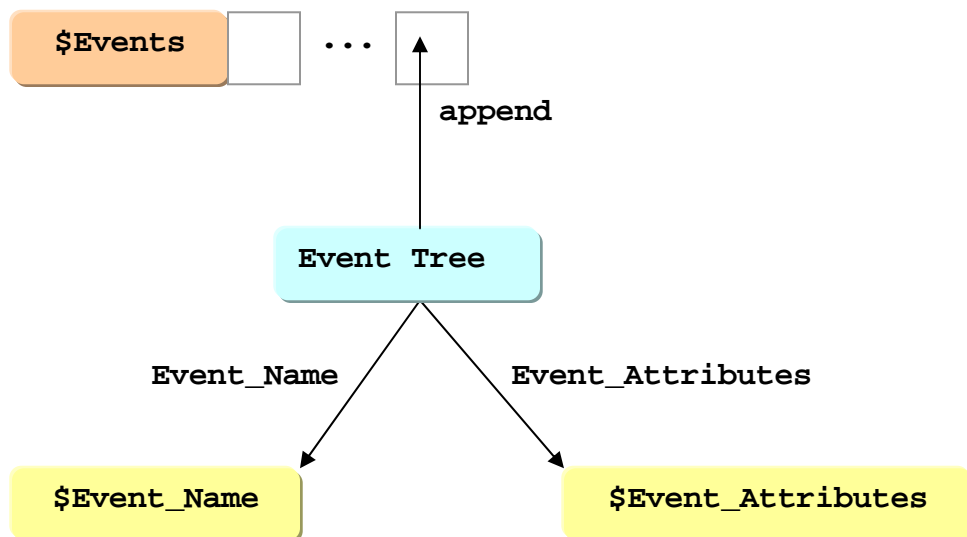


Figure 67. The **#Event** Parsed Tree Structure.

b. *The #Event_Attributes Rule*

The event rule expects the input token stream to contain the pattern shown in Figure 68 below. The **#Event** rule is derived from METALS grammar Rule 6.

```
[ $Var := ( 'STO'!'DET' ) ]  
[ 'MEAN' '=' $Mean ]  
[ 'DUR' '=' $Duration ]  
[ $Additional_Attribs := #CPP_Code ]
```

Figure 68. Expected Input Pattern for **#Event_Attributes**.

The rule expects attributes to be specified using the keywords **STO**, **DET**, **MEAN**, **DUR**. Additional attributes are also expected and identified by calling the **#CPP_Code** rule to collect a stream of text representing source code written in the target language in a list as illustrated in Figure 69 below:

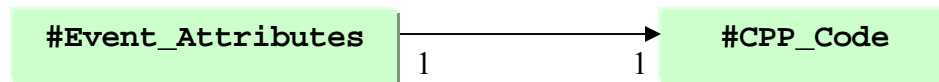


Figure 69. Program Execution Flow for **#Event_Attributes** Rule.

The resulting output returned by the **#Event_Attributes** rule is in the form of a tree whose structure is shown in Figure 70. This tree data structure is then assigned to the **\$Event_Attributes** variable in the **#Event** rule.

```
<. Var: $Var,  
  Mean: $Mean,  
  Duration: $Duration,  
  Additional_Attribs: $Additional_Attribs .>
```

Figure 70. Output returned by the **#Event_Attributes** rule.

The equivalent visual notion for the parsed tree is shown in Figure 71 below:

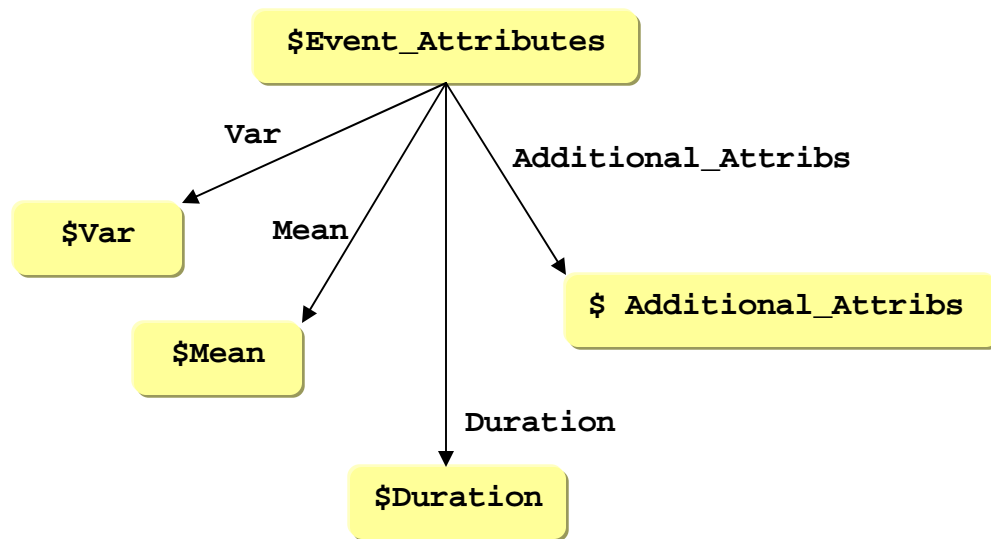


Figure 71. The **#Event_Attributes** Parsed Tree Structure.

6. The **#Chain** and **#Rule** Rules

a. The **#Chain** Rule

The chain rule expects the input token stream to contain the pattern shown in Figure 72 below. The **#Chain** rule is derived from METALS grammar Rule 7.

```
'CHAIN' $Chain_Name
'{ (* $Rules !.:= #Rule *) }'
```

Figure 72 Expected Input Pattern for **#Event**.

The rule expects to detect the use of the keyword **CHAIN** followed by an identifier representing the name of the chain object and finally a list of event rules associated with this chain. To obtain the events rules, the rule **#Rule** is called to return each rule detected in the input stream. This call is illustrated in Figure 73 below:

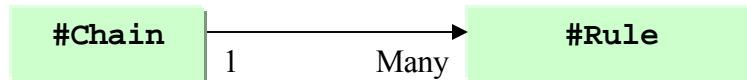


Figure 73. Program Execution Flow for **#Chain** rule.

The resulting output returned by the **#Chain** rule is in the form of a tree whose structure is shown in Figure 74. This tree data structure is then inserted into the **\$Chains** list in the main **#Parse** rule.

```

<. Chain_Name: $Chain_Name,
  Rules: $Rules .>
  
```

Figure 74. Output returned by the **#Chain** rule.

The equivalent visual notion for the parsed tree is shown in Figure 75 below:

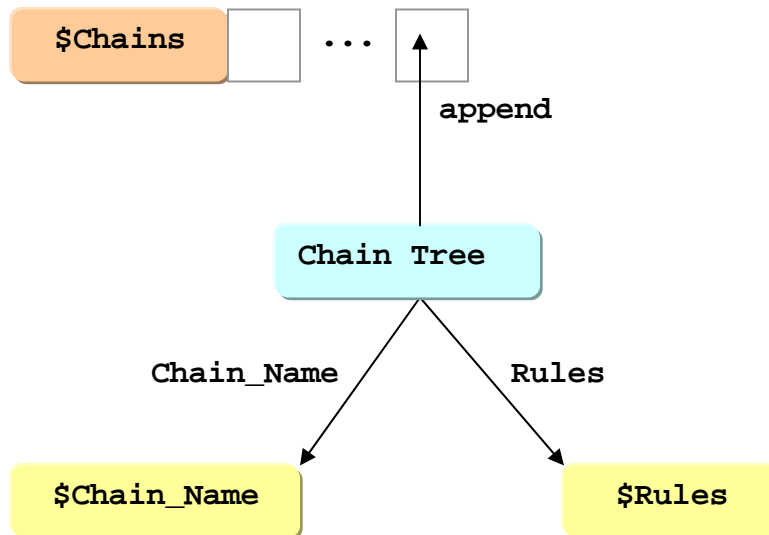


Figure 75. The **#Chain** Parsed Tree Structure.

b. The #Rule Rule

This rule expects the input token stream to contain the pattern shown in Figure 76 below. The **#Rule** rule is derived from METALS grammar Rule 7C specifically.

```
$Rule_Name  
[':' ( * $Patterns !.:= #Pattern * ) ] ';' .
```

Figure 76. Expected Input Pattern for **#Rule**.

The rule uses no keyword but first expects an identifier representing the name of the rule before the special **:** symbol. A list of event patterns is expected after this symbol optionally. The rule **#Pattern** is used to match incoming streams of tokens to a list of pre-defined standard event patterns. This call is illustrated in Figure 77 below:

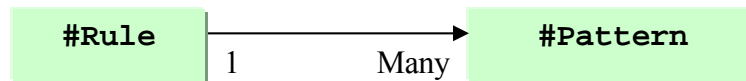


Figure 77. Program Execution Flow for **#Rule** rule.

The resulting output returned by the **#Rule** rule is in the form of a tree whose structure is shown in Figure 78. This tree data structure is then inserted into the **\$Rules** list in the **#Chain** rule.

```
<. Rule_Name: $Rule_Name,  
Patterns: $Patterns,  
Used_Event_Names: $used_event_names,  
number_of_occurences: $num_of_occurences .>
```

Figure 78. Output returned by the **#Chain** rule.

The equivalent visual notion for the parsed tree is shown in Figure 79 below:

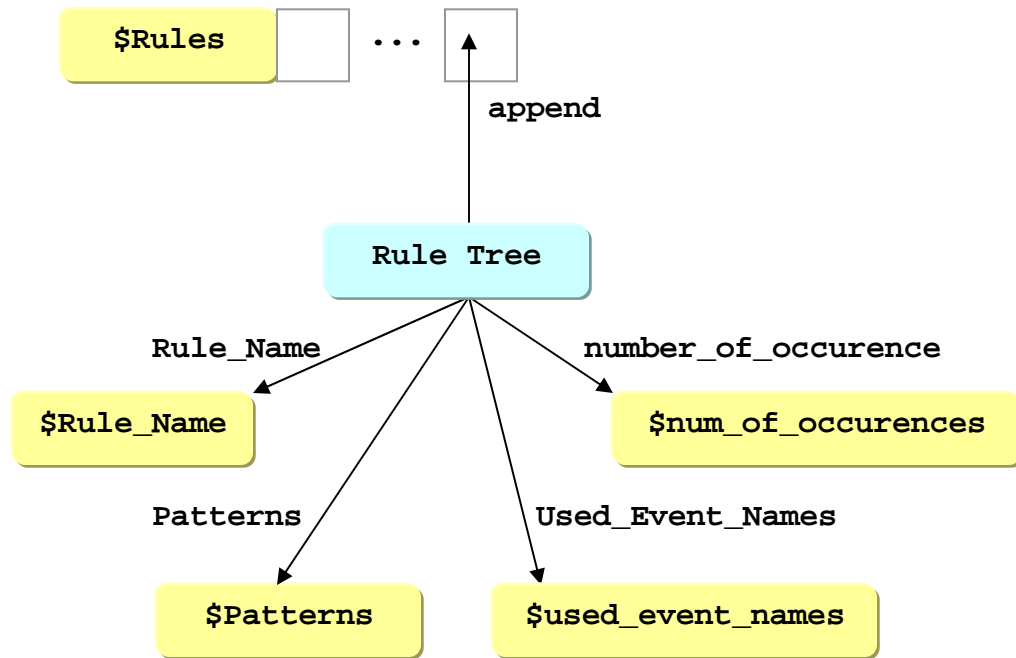


Figure 79. The **#Rule** Parsed Tree Structure.

7. Rules used to match and identify Event Patterns

a. The **#Pattern** Rule

The pattern rule is a recursive rule that is either called to pattern match and identify event patterns in the input stream. It is derived from METALS grammar Rule 8. This rule demonstrates the RIGAL language's ability to perform a top-down recursive parsing in the traditional LL(1) sense but with the addition ability and flexibility to optionally perform backtracking in the event of a parse failure. The backtracking feature is supported in RIGAL by the ability to define alternative patterns within a single rule. Whenever an input token stream fails to match one pattern inside a rule, the same input token stream will be read again and compared with the other alternative patterns that has been defined in the rule. A failure will cause the parser to backtrack and read the same input token stream again.

The `#Pattern` rule contains two alternative patterns that is used to match in the incoming token stream. The first attempts to match the incoming token stream with 7 pre-defined standard event types prescribed by the METALS language, while the second one attempts to detect a group of patterns enclosed within braces. The two alternative patterns in the rule is separated by the special `;;` symbol. The expected input patterns are shown in Figure 80 below:

```
( #Iteration !  
  #Loop !  
  #Conditional !  
  #Alternative !  
  #Action !  
  #Group !  
  #Simple )  
;;  
'(' (* $List !.:= #Pattern *) ')'
```

Figure 80. The Expected Input Pattern for `#Pattern`.

A specific rule is called for each of the standard event patterns, and the order in which these rules are called is sequential. For example, the parser will first attempt to identify an iteration event pattern failing which it will then proceed to identify a loop event pattern and so forth with the same input token stream.

The two alternative patterns defined in the `#Pattern` rule means that the rule will have two possible outputs. For the first standard event pattern matching alternative, the rule will return a tree with the structure shown in Figure 83.

```
<. Type: $Type, Body: $Pattern .>
```

Figure 81. Output for a single event pattern.

For the second list of patterns, alternatively the rule will return a list of patterns shown below in Figure 82, each with a structure described in Figure 84.

\$List

Figure 82. Output for a list of event patterns.

The output tree or list data structure is then inserted into the **\$Patterns** list in the **#Rule** rule. The equivalent visual notions for two possible outputs are shown in Figure 83 and 84 below:

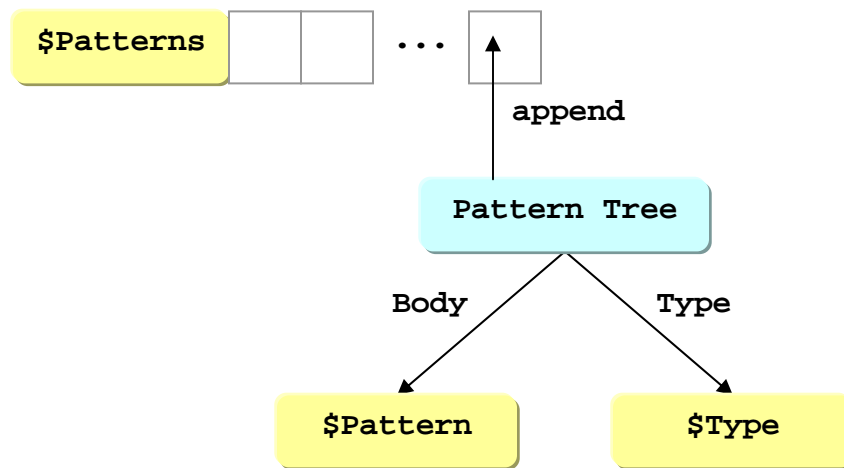


Figure 83. The **#Pattern** Parsed Tree Structure for A Single Pattern.

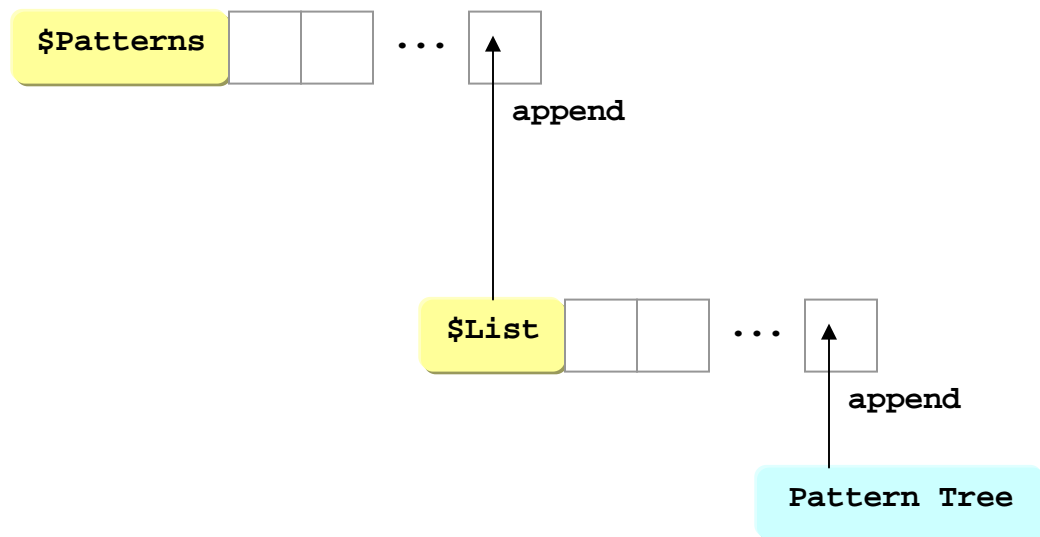


Figure 84. The **#Pattern** Parsed Tree Structure for Lists of Patterns.

b. The #Iteration Rule

The **#Iteration** rule is derived from the METALS grammar Rule 8I. The rule expects the input token stream to contain the pattern shown in Figure 85 below.

```
'REPEAT'  
' ('  
$Op := #Operator  
$Expression := #Plain_Text  
)'  
'{ ' (* $Pattern !.:= #Pattern *) '}'
```

Figure 85. Expected Input Pattern for **#Iteration**.

The rule expects to detect the use of the keyword **REPEAT** followed by a combination of an operator and an expression in plain text before proceeding to identify a list of patterns to be collected. This rule calls the **#Operator**, **#Plain_Text** and the recursive **#Pattern** rules to obtain the specified operator, expression and list of patterns respectively.

The resulting output returned by the **#Iteration** rule is in the form of a tree whose structure is shown in Figure 86. This tree data structure is then assigned to the **\$Pattern** variable in the **#Pattern** rule.

```
<. Repeat : ( . $Op $Expression . ),  
  This: $Pattern .>
```

Figure 86. Output returned by the **#Iteration** rule.

The equivalent visual notion for the parsed tree is shown in Figure 87 below:

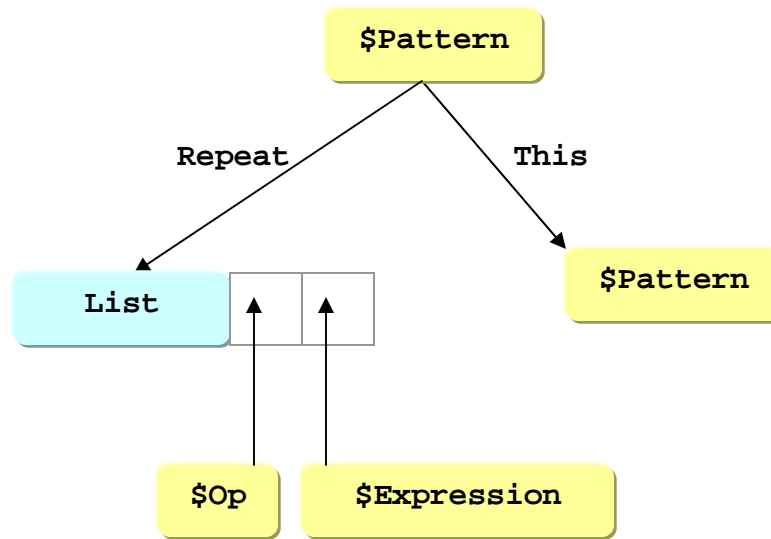


Figure 87. The **#Iteration** Parsed Tree Structure.

*c. The **#Loop** Rule*

The **#Loop** rule is derived from the METALS grammar Rule 8H. The rule expects the input token stream to contain the pattern shown in Figure 88 below.

```

'WHILE'
'(' $Bool := #Bool_Expression ')'
'{' (* $Pattern !.:= #Pattern *) '}'

```

Figure 88. Expected Input Pattern for **#Loop**.

The rule expects to detect the use of the keyword **WHILE** followed by a Boolean expression before proceeding to identify a list of patterns to be collected. This rule calls the **#Bool_Expression** and the recursive **#Pattern** rules to obtain the specified Boolean expression and list of patterns respectively.

The resulting output returned by the **#Loop** rule is in the form of a tree whose structure is shown in Figure 89. This tree data structure is then assigned to the

`$Pattern` variable in the `#Pattern` rule.

```
<. While : $Bool,  
  Do : $Pattern .>
```

Figure 89. Output returned by the `#Loop` rule.

The equivalent visual notion for the parsed tree is shown in Figure 90 below:

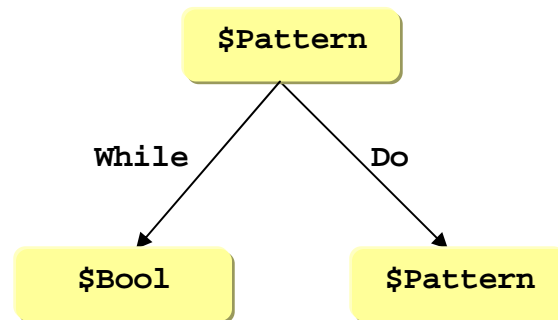


Figure 90. The `#Loop` Parsed Tree Structure.

d. The `#Conditional` Rule

The `#Conditional` rule is derived from the METALS grammar Rule 8D.

The rule expects the input token stream to contain the pattern shown in Figure 91 below.

```
'WHEN'  
'(' $Bool := #Bool_Expression ')'  
'{' (* $Pattern !.:= #Pattern *) '}'  
[ 'ELSE'  
'{' (* $Pattern2 !.:= #Pattern *) '}' ]
```

Figure 91. Expected Input Pattern for `#Conditional`.

The rule expects to detect the use of the keyword `WHEN` followed by a Boolean expression before proceeding to identify a list of patterns to be collected. The rule also expects the optional use of the keyword `ELSE` and a list of alternative event patterns to be collected. This rule calls the `#Bool_Expression` and the recursive

#Pattern rules to obtain the specified Boolean expression and the lists of patterns respectively.

The resulting output returned by the **#Conditional** rule is in the form of a tree whose structure is shown in Figure 92. This tree data structure is then assigned to the **\$Pattern** variable in the **#Pattern** rule.

```
<. When: $Bool,  
    Do: $Pattern,  
    Else: $Pattern2 .>
```

Figure 92. Output returned by the **#Conditional** rule.

The equivalent visual notion for the parsed tree is shown in Figure 93 below:

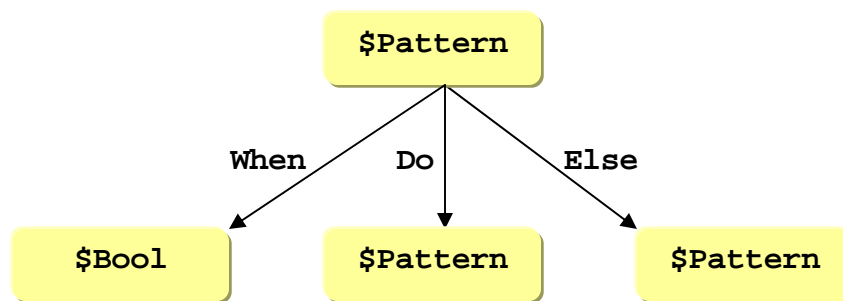


Figure 93. The **#Conditional** Parsed Tree Structure.

*e. The **#Alternative** Rule*

The **#Alternative** rule is derived from the METALS grammar Rule 8E. The rule expects the input token stream to contain the pattern shown in Figure 94 below.

```
'DECIDE'  
'(' (* $Outcome := #Outcome * '|' ) ')'
```

Figure 94. Expected Input Pattern for **#Conditional**.

The rule expects to detect the use of the keyword **DECIDE** followed by a series of alternative outcomes separated by the **|** special symbol. This rule calls the **#Outcome** rule to obtain an outcome which is either an event or a series of events with a probability value associated with it.

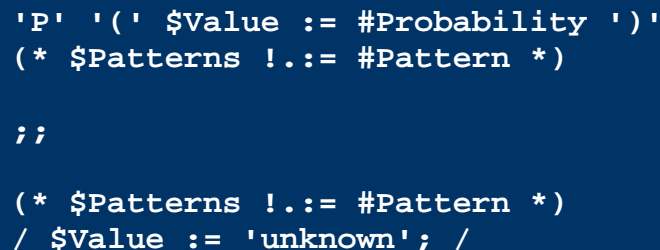
The resulting output returned by the **#Alternative** rule is in the form of a list of outcomes as shown in Figure 95. This list data structure is then assigned to the **\$Pattern** variable in the **#Pattern** rule.



```
$List_Of_Outcomes
```

Figure 95. Output returned by the **#Alternative** rule.

The **#Outcome** rule that is called by the **#Alternative** rule is derived from the METALS grammar Rule 8F. The rule contains two alternative patterns shown in Figure 96 below.



```
'P' '(' $Value := #Probability ')'
(* $Patterns !.:= #Pattern *)

;;

(* $Patterns !.:= #Pattern *)
/ $Value := 'unknown'; /
```

Figure 96. Expected Input Pattern for **#Outcome**.

The rule either expects a probability value to be associated with a list of event patterns using the **P(Value)** constructor or no associated at all. Should the constructor be detected, the rule will call the **#Probability** rule to obtain a probability value. The reason why an alternative pattern that allows no probability values to be associated with an outcome is to provide flexibility and versatility. For example, the code generator can assign equal probabilities to all outcomes should none be explicitly defined.

The **#Probability** rule that is called by the **#Outcome** rule is derived from the METALS grammar Rule 8G. The rule essential collects tokens and concatenate (using the RIGAL built-in rule **#IMPLode**) them into a string of characters that is supposed to represent a float number as shown in Figure 97 below:

```
(* $List !.:= S'($$ <> ' ')* )  
/ IF ($List[1] = '.') ->  
  $List := (. 0 .)!!$List; FI;  
  $Value := #IMPLode($List);
```

Figure 97. Expected Input Pattern for **#Outcome**.

The rule allows probability values to be expressed in the shorter dot-value format (for example **.8**) rather than the standard zero-dot-value format (for example **0.8**).

The resulting outputs returned by the rules **#Probability**, **#Outcome** and **#Alternative** are shown in Figures 98, 99 and 100 below respectively.

```
$Value
```

Figure 98. Output returned by the **#Probability** rule.

```
<. Patterns: $Patterns,  
  Probability: $Value .>
```

Figure 99. Output returned by the **#Outcome** rule.

```
$List_Of_Outcomes
```

Figure 100. Output returned by the **#Alternative** rule.

The equivalent visual notion for the parsed tree is shown in Figure 101

below:

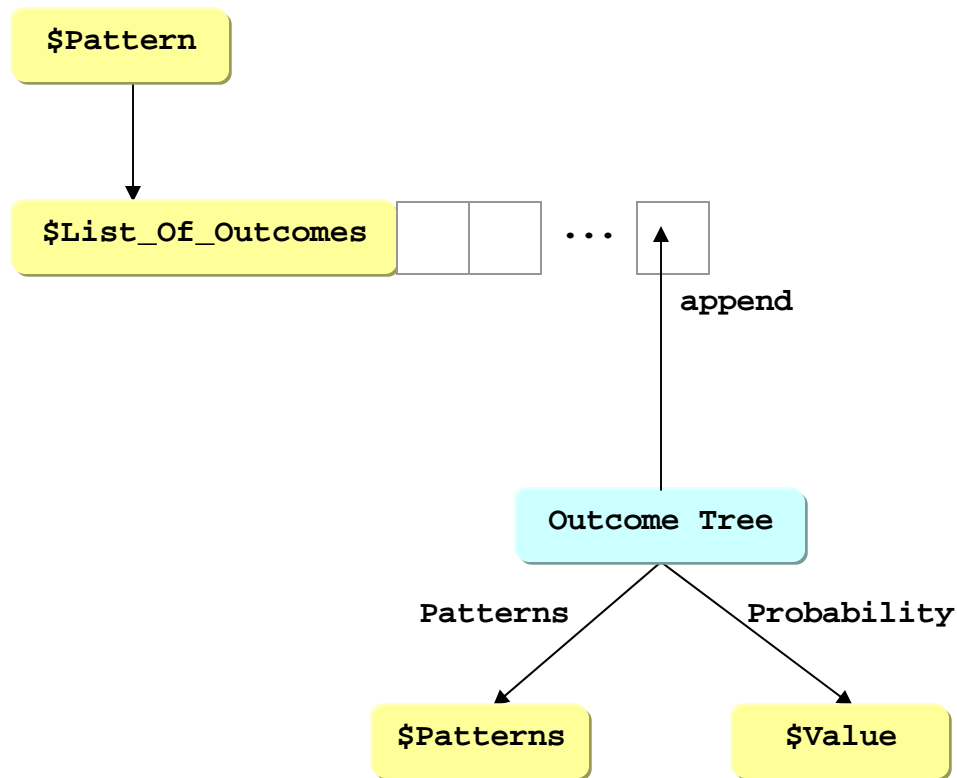


Figure 101. The **#Alternative** Parsed Tree Structure.

f. The #Simple Rule

The **#Simple** rule is derived from the METALS grammar Rule 8B. The rule expects the input token stream to contain the pattern shown in Figure 102 below.

```

$Id
/
LAST #Rule $used_event_names ++:=
<. $Id:T .>;

LAST #Rule $num_occurences ++:=
<. $Id: LAST #Rule $num_occurences.$Id +1 .>;
/

```

Figure 102. Expected Input Pattern for **#Simple**.

The rule expects only an identifier **\$Id** and will return it in a tree together with the updated value of the variable **\$num_occurences** with is declared as global to all rules within the **#Rule** rule. The **\$num_occurences** value will be used by the code generator to uniquely identify the current instance of this event should multiple instances of this event exists within a rule. This tree data structure shown in Figure 104 below is then assigned to the **\$Pattern** variable in the **#Pattern** rule.

```
<. Event_Name: $Id,  
  order: LAST #Rule $num_occurences.$Id - 1 .>
```

Figure 103. Output returned by the **#Simple** rule.

The equivalent visual notion for the parsed tree is shown in Figure 101 below:

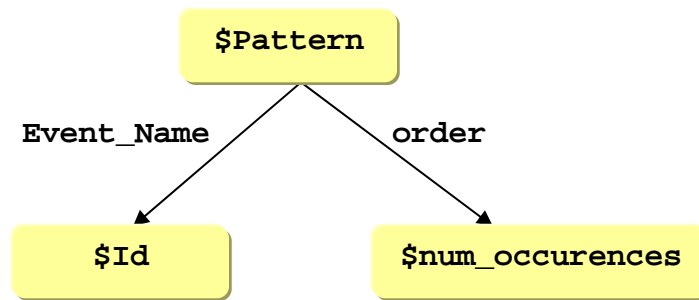


Figure 104. The **#Simple** Parsed Tree Structure.

g. The **#Group** Rule

The **#Group** rule is derived from the METALS grammar Rule 8J. The rule expects the input token stream to contain the pattern shown in Figure 105 below.

```
'(' (* $Group_Of_Patterns !.:= #Pattern *) ')'
```

Figure 105. Expected Input Pattern for **#Group**.

The rule is used to allow any arbitrary group of event patterns to be grouped by enclosing them within brackets.

The resulting output returned by the **#Group** rule is in the form of a list of patterns as shown in Figure 106. This list data structure is then assigned to the **\$Pattern** variable in the **#Pattern** rule.

```
$Group_Of_Patterns
```

Figure 106. Output returned by the **#Alternative** rule.

The equivalent visual notion for the parsed tree is shown in Figure 107 below:

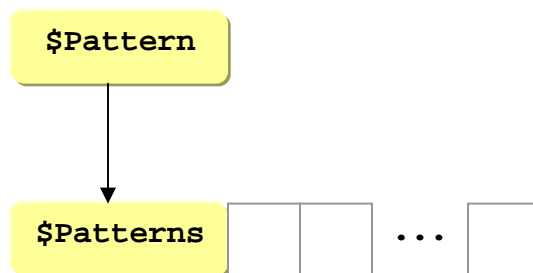


Figure 107. The **#Group** Parsed Tree Structure.

*h. The **#Action** Rule*

The **#Action** rule is derived from the METALS grammar Rule 8C. The rule is used to collect a stream of input tokens and convert them into text strings using the **#CPP_Code** rule as shown by the RIGAL in Figure 108. The output text strings are code written in the target language by the user that is in a form that is readable by the code generator.

```
$Action := #CPP_Code
```

Figure 108. Expected Input Pattern for **#Group**.

The resulting output returned by the `#Action` rule is in the form of a list of character tokens (representing source code) as shown in Figure 109. This list data structure is then assigned to the `$Pattern` variable in the `#Pattern` rule.



Figure 109. Output returned by the `#Action` rule.

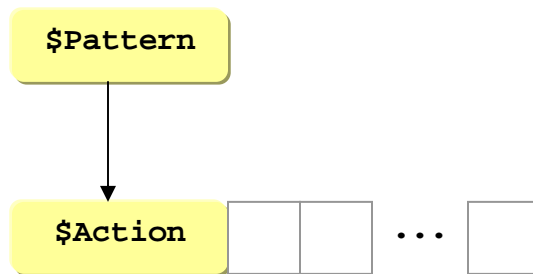


Figure 110. The `#Action` Parsed Tree Structure.

8. Program Execution Flow For Rules Dealing With Event Patterns

The program execution flow for rules dealing with event patterns is illustrated in Figure 111. During each call to the `#Pattern` rule, each of the 7 rules representing the seven standard event patterns will be called in order until a pattern match is successful. With the exception of the `#Simple` and `#Action` rule, the other 5 event pattern matching rules, namely the `#Iteration`, `#Loop`, `#Conditional`, `#Alternative` and `#Group` rules will call the `#Pattern` rule recursively.

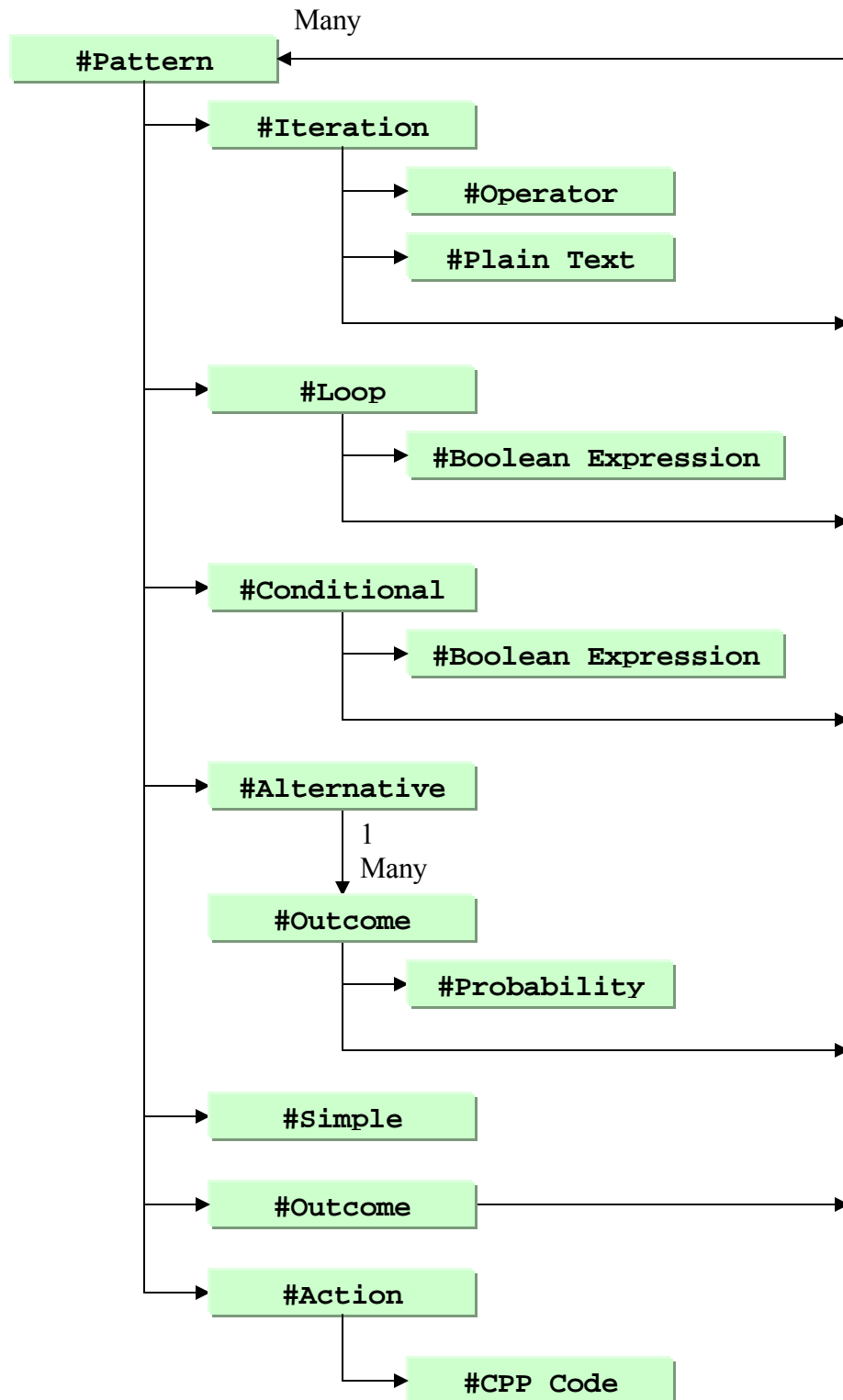


Figure 111. Program Execution Flow for **#Pattern**.

9. The String Processing Rules

In addition to rules described earlier, the METALS parser also include special string processing rules to ensure that the strings generated by the rules in the parser can be interpreted correctly by the code generator. These rules include:

- The **#Operator** Rule - The RIGAL lexical analyzer will treat an string input of the logical operator `<=` as two separate token `<` and `=`. This rule is necessary to correct this by concatenating them together again. This rule can be expanded for all other similar requirements of concatenating two tokens as necessary. Currently it is only used by the **#Iteration** rule.
- The **#Boolean_Expression** Rule - This rule is used to collect all tokens enclosed between the keywords `BEGIN` and `END` inside a list that will represent a Boolean expression written in the target language that is to be output directly by the code generator without modifications. Currently it is used by the **#Loop** and **#Conditional** rules.
- The **#Plain_Text** Rule - This rule is used to collect all tokens enclosed within curly braces `{` and `}` inside a list that will represent a section of plain text to be output directly by the code generator without modifications. Currently it is only used by the **#Iteration** rule.
- The **#CPP_Code** Rule - This rule is used to collect all tokens enclosed between the keywords `BEGIN` and `END` inside a list that will represent code written in the target language that are to be output directly by the code generator without modifications. In addition, the rule will also concatenate the pairs of tokens into standard symbols required in the target language (C++) listed in Table 1 below. Currently it is only used by the **#Action** rule.

Input Token Pair		Output Single Token
/	/	//
<	=	<=
>	=	>=
=	=	==
!	=	!=
+	+	++
-	-	--
+	=	+=
-	=	-=
&	&	&&
<	<	<<
>	>	>>

Table 2. Token pair concatenation table.

E. THE METALS CODE GENERATOR

1. The Main #Generate Rule

The METALS code generator is built using a set of RIGAL rules based on the syntax of the target language which is ANSI C++ for this current implementation. The code generator receives an input parsed tree of the METALS source code from the parser and uses the main #Generate Rule to generate the equivalent C++ simulation program.

The expected input pattern for the code generator is the parsed tree returned by the parser followed by the name of the input METALS source file shown in Figure 112 below:

```
<. Title: $Title,  
  Headers: $Headers,  
  Worlds: $Worlds,  
  Entities: $Entities,  
  Events: $Events,  
  Chains: $Chains .>  
$Filename
```

Figure 112. The METALS Parsed Tree Structure.

Code is generated by the parsed tree into 5 C++ source files that together constitutes the C++ simulation program. The structure of this program is shown in Figure 113 below:

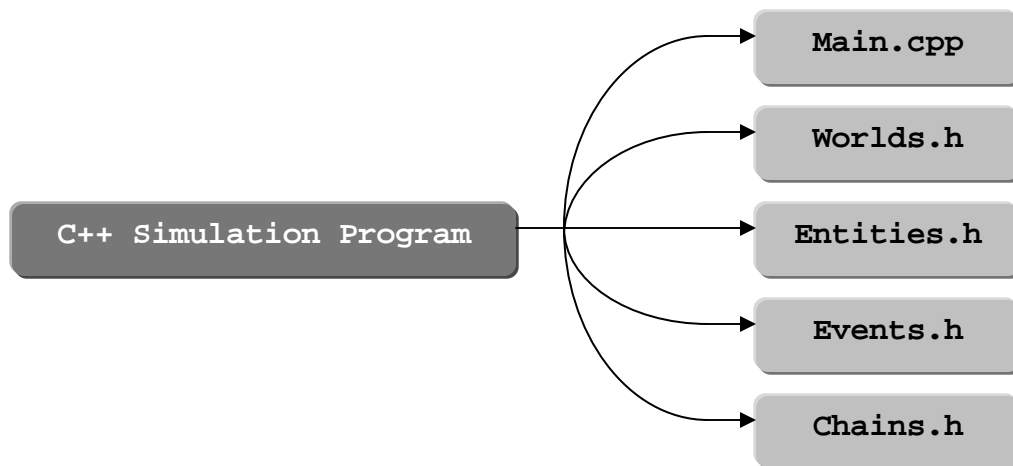


Figure 113. Structure of generated C++ simulation program.

Each of the source files is opened for writing using the OPEN file_object filename built-in RIGAL constructor analogous to the file I/O object in C++. The code fragment that opened the 5 output files for writing is shown here in Figure 114:

```
OPEN GEN_MAIN 'Main.cpp';  
OPEN GEN_WORLD 'Worlds.h';  
OPEN GEN_ENTITIES 'Entities.h';  
OPEN GEN_EVENTS 'Events.h';  
OPEN GEN_CHAINS 'Chains.h';
```

Figure 114. Opening 5 output files for writing.

The C++ code generation sequence follows the general structure of a C++ program, in other words inclusion of headers files → class definitions → function prototypes → functions / sub-routines → C++ main routine.

Each section of the C++ simulation program is generated by feeding each branch of the input parsed tree into separate code generating rules as shown in Figure 115 below:

```
#Generate_Title($Title $Filename );  
#Generate-Headers($Headers);  
#Generate_Worlds($Worlds);  
#Generate_Entities($Entities);  
#Generate_EventClasses($Events);  
#Generate_EventChains($Chains);
```

Figure 115. Feeding the parsed tree into specific code generating rules.

Text is written directly to a file using the << operator similar to output stream operator in C++. In the **#Generator** rule, the main routine for the C++ simulation is created. A simplified version of the series of statements necessary to accomplish this is shown in Figure 116. The starting point for the simulation is the first rule that the code generator obtains from first event chain in the list of chains **\$Chains**.

```

-- Main Program
GEN_MAIN << 'int main(int argc, char *argv[]);
GEN_MAIN << '{';
GEN_MAIN << $First_Rule.Rule_Name '_class '
             $First_Rule.Rule_Name ';';
GEN_MAIN << $First_Rule.Rule_Name '_rule'
             $First_Rule.Rule_Name ' );';
GEN_MAIN << 'system("PAUSE");';
GEN_MAIN << 'return 0;';
GEN_MAIN << '};

```

Figure 116. Generating the C++ main program.

2. The Code Generation Template

a. The `#Generate_Title` Rule

This rule receives and processes the branch `Title` from the overall parsed tree followed by the METALS input source file name as shown in Figure 117 below:

```

<. Title: $Title,
   Description: $Description .>
$Fname

```

Figure 117. Input to `#Generate_Title` Rule.

Using this information, the rules generates the output C++ code in the `Main.Cpp` file shown in Figure 118 below:

```

// METALS Code Generator Version 0.4C
// C++ simulation program [$Title] created from $Fname
// $Description

```

Figure 118. Output from `#Generate_Title` Rule.

b. The #Generate-Headers Rule

This rule receives and processes the branch **Headers** from the overall parsed tree which is a simple list of user defined header file names shown in Figure 119 below:



\$Headers

Figure 119. Input to **#Generate-Headers** Rule.

Using this information, the rules generates the output C++ code in the **Main.Cpp** file shown in Figure 120 below. The header files that will be generated by the code generators, namely **Worlds.h**, **Entities.h**, **Events.h** and **Chains.h** are all included in the output source code.

```
#include <iostream>
#include <ctime>;
#include <cstdlib>;
#include <stdlib.h>;

using namespace std;;

#include "Worlds.h";
#include "Entities.h";

#include $Headers[1]
.
.
.
#include $Headers[n] // for n number of header files.

#include "Events.h";
#include "Chains.h";
```

Figure 120. Output from **#Generate-Headers** Rule.

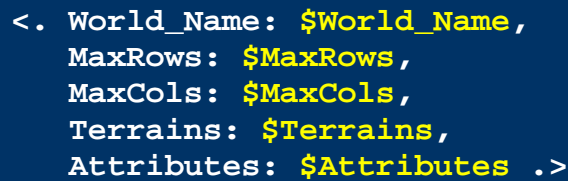
c. *The #Generate_Worlds Rule*

This rule receives and processes the branch `Worlds` from the overall parsed tree which is a list of user defined world objects shown in Figure 121 representing the environment within which entities reside. Each of these world objects in the list `$Worlds` has a data structure shown in Figure 122 below:



`$Worlds`

Figure 121. Input to `#Generate_Worlds` Rule.



```
<. World_Name: $World_Name,
  MaxRows: $MaxRows,
  MaxCols: $MaxCols,
  Terrains: $Terrains,
  Attributes: $Attributes .>
```

Figure 122. Data structure for each world object in list `$Worlds`.

Using this information, for each of the world object that the user has defined, the rule generates the output C++ code in the `Worlds.h` header file shown in Figure 123. The branch `Terrains` contains a list of user defined terrain types for the environment, while the branch `Attributes` contains additional attributes that the user has defined. The generated source code contains class definitions for a world object as well as an instantiation of this object at the end.

```

// METALS Code Generator Version 0.4C
// Generation of Worlds

__int64 $World_Name_MaxRows = $MaxRows;
__int64 $World_Name_MaxCols = $MaxCols;

enum $World_Name_Terrain { $Terrains[1] . $Terrains[n] }

class $World_Name_class
{
    public:
        $World_Name_class();

        __int64 BeenHere;';
        bool Marked;';
        $World_Name_Terrain ObjectType;

        $Attributes;
};

$World_Name_class :: $World_Name_class()
{
    BeenHere = 0;
    Marked = false;
    ObjectType = Empty;
}

$World_Name_class
$World_Name [$World_Name_MaxRows][ $World_Name_MaxCols];

```

Figure 123. Output for each world object from `#Generate_Worlds` Rule in `Worlds.h`.

d. The #Generate_Entities Rule

This rule receives and processes the branch `Entities` from the overall parsed tree which is a list of user defined entity objects shown in Figure 124 representing the entities participating in the simulation. Each of these entity objects in the list `$Entities` has a data structure shown in Figure 125 below:

```
$Entities
```

Figure 124. Input to `#Generate_Entities` Rule.

```
<. Entity_Name: $Entity_Name,  
   Entity_Attributes: $Entity_Attributes .>
```

Figure 125. Data structure for each entity object in list `$Entities`.

Using this information, for each of the entity object that the user has defined, the rule generates the output C++ code in the `Entities.h` header file shown in Figure 126. The branch `Entity_Attributes` contains additional attributes that the user has defined. The generated source code contains class definitions for an entity object as well as an instantiation of this object at the end.

```
// METALS Code Generator Version 0.4C  
// Generation of Entities  
  
class $Entity_Name_class  
{  
    public:  
        $Entity_Name_class();  
        $Entity_Attributes;  
}  
  
$Entity_Name_class $Entity_Name;
```

Figure 126. Output from `#Generate_Entities` Rule in `Entities.h`.

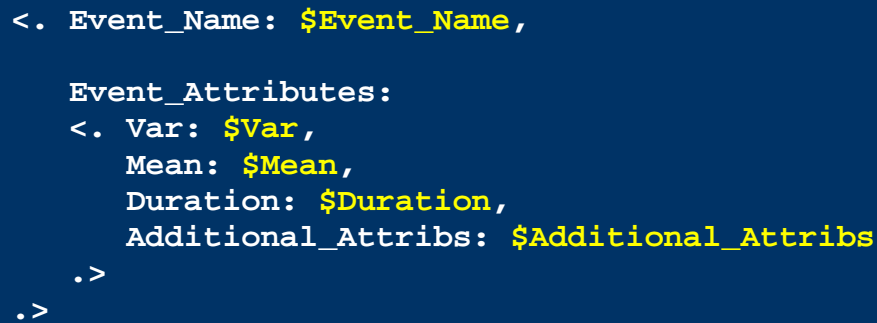
e. *The #Generate_EventClasses Rule*

This rule receives and processes the branch **Events** from the overall parsed tree which is a list of user defined event objects shown in Figure 127 representing the events that will occur in the simulation. Each of these event objects in the list **\$Events** has a data structure shown in Figure 128 below:



\$Events

Figure 127. Input to **#Generate_EventClasses** Rule.



```
<. Event_Name: $Event_Name,

  Event_Attributes:
    <. Var: $Var,
      Mean: $Mean,
      Duration: $Duration,
      Additional_Attribs: $Additional_Attribs
    .>
.>
```

Figure 128. Data structure for each event object in list **\$Events**.

Using this information, for each of the n number of event objects that the user has defined in the **\$Events** list, the rule generates the output C++ code in the **Events.h** header file shown in Figure 129. A static member **Instances** is created for each event and incremented by the event class constructor to keep track of the total number of instantiations for this event class. A discrete Poisson random number generator is used to determine the duration if the event attribute Var is set as STO (Stochastic). A global event count variable is also declared and incremented in the constructor to keep track of the total number of events regardless of its type.


```

// METALS Code Generator Version 0.4C';
// Generation of Event Classes';

long Global_Event_Count;
enum EType { DET, STO };

class $Events[n].Event_Name_class
{
    public:
        $Events[n].Event_Name_class();
        string Name;
        EType Var;
        double Mean;
        double Duration;
        $Event[n].Event_Attributes.Additional_Attribs
        static long Instances;
};

long $Events[n].Event_Name_class :: Instances = 0;

$Events[n].Event_Name_class ::
$Events[n].Event_Name_class()
{
    Global_Event_Count++;
    Instances++;
    Name = $Events[n].Event_Name;
    Var = $Events[n].Event_Attributes.Var;
    Mean = $Events[n].Event_Attributes.Mean;
    Duration = $Events[n].Event_Attributes.Duration;
    if (Var==STO) // Discrete Poisson RNG.
        double a = exp(-Mean);
        long p = 1; long x = 1;
        while (p > a) {
            double U = (double) rand() / RAND_MAX;
            p = p*U;
            x++; }
        Duration = x;
}

```

Figure 129. Output from `#Generate_Events` Rule in `Events.h`.

*f. The **#Generate_EventChains** and related rules*

Each event chain comprises a number event rules. Each event rule comprises a number of events in order of their occurrence. Each event has a particular pattern of behavior. Due to this hierarchical data structure of the event chain, a hierarchy of rules are used to generate specific sections of code needed to generate each layer in event chain. The top-down mapping between the event chain data structure and the code generator rule calls is illustrated in Figure 130.

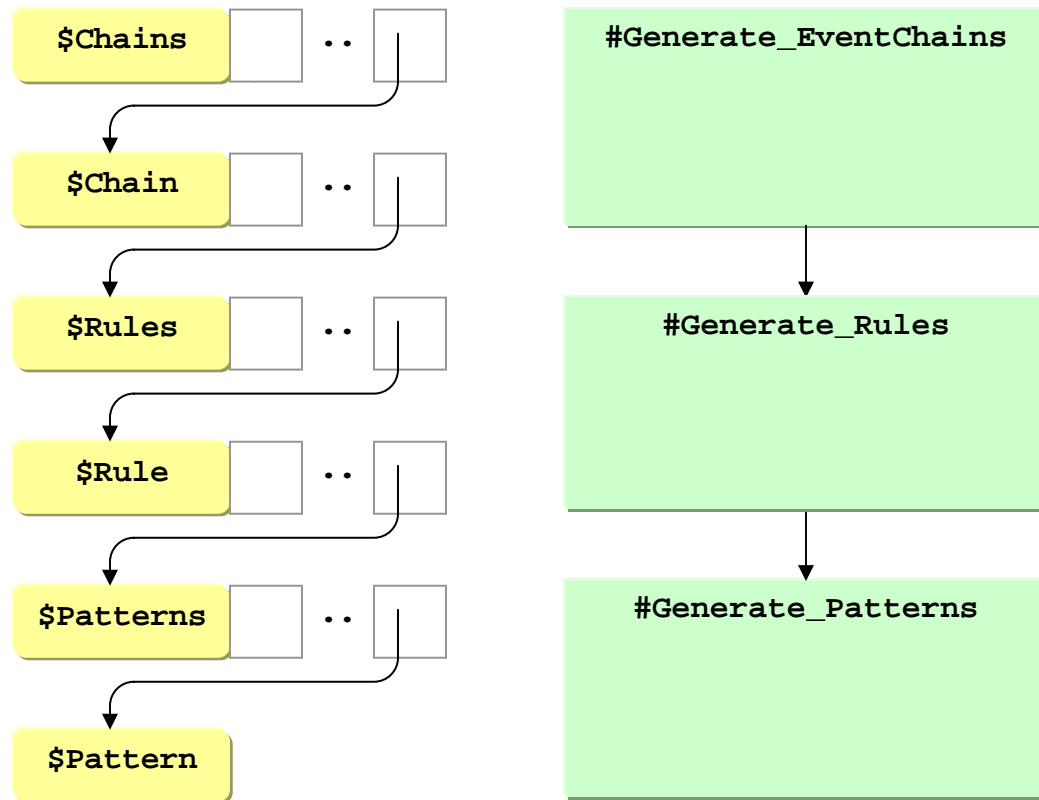


Figure 130. Event chain code generation mapping.

The topmost **#Generate_EventChains** rule receives and processes the branch **Chains** from the overall parsed tree which is a list of user defined chains of events that constitute the simulation shown in Figure 131.

```
$Chains
```

Figure 131. Input to `#Generate_EventChains` Rule.

Each event chain in `$Chains` has the data structure shown in Figure 132 below:

```
<. Chain_Name: $Chain_Name,  
  Rules: $Rules .>
```

Figure 132. The input data structure of an event chain.

The name of the chain and a list of rules `$Rules` are used by the code generator to generate the C++ code in the `Chains.h` header file shown in Figure 133. The list of rules are used to generate the C++ functional prototypes and then passed to the `#Generate_Rules` rule to generate lower level code fragments.

```
// METALS Code Generator Version 0.4C  
// Generation of Event Chains  
  
// User defined chain $Chain_Name  
  
// Functional prototypes for n number of rules  
// in one event chain  
$Rules[1].Rule_Name_class  
$Rules[1].Rule_Name_rule( $Rule.Rule_Name_class );  
.  
.  
$Rules[n].Rule_Name_class  
$Rules[n].Rule_Name_rule( $Rule.Rule_Name_class );  
  
// Specific code returned by RIGAL rule  
// #Generate_Patterns for n number of rules  
#Generate_Rules($Rule[1]);  
.  
.  
#Generate_Rules($Rule[n]);
```

Figure 133. Code generated directly by `#Generate_EventChains`.

Each rule in `$Rules` has the data structure shown in Figure 134 below:

```
<. Rule_Name: $Rule_Name,  
  Patterns: $Patterns,  
  Used_Event_Names: $used_event_names,  
  number_of_occurrences: $num_of_occurrences .>
```

Figure 134. The input data structure of an event chain.

The name of the rule and a list of event patterns `$Patterns` are used by the code generator to generate the C++ code in the `Chains.h` header file shown in Figure 135. The names of all events that are in the event chain are stored in the list `$used_event_names`. An instance for each event object in this list is first created before `#Generate_Patterns` rule is called to generate the specific code associated with each event object. An instance of the rule is also created and returned to allow event attributes to be manipulated and propagated externally.

```
$Rule_Name_class $Rule_Name_rule( $Rule_Name_class )  
{  
    $Rule_Name_class $Rule_Name;  
    // Functional prototypes for n number of events  
    $used_event_names[1]_class  
    $used_event_Names[1]  
    [$num_of_occurrences.$used_event_names[1]]  
    .  
    .  
    $used_event_names[n]_class  
    $used_event_Names[n]  
    [$num_of_occurrences.$used_event_names[n]]  
  
    // Specific code returned by RIGAL rule  
    // #Generate_Patterns for n number of event patterns  
    #Generate_Patterns($Patterns[1])  
    .  
    .  
    #Generate_Patterns($Patterns[n])  
    return $Rule_Name;  
}
```

Figure 135. Code generated directly by `#Generate_Rules`.

Depending on what type of standard event pattern that is encountered, the `#Generate_Patterns` rule will generate the corresponding equivalent C++ code in the `Chains.h` header file. Each event pattern has a data structure shown in Figure 136 below. The body `$Pattern` for each event is passed into a specific code generating rule that has been developed based on its type specified by `$Type`.

```
<. Type: $Type, Body: $Pattern .>
```

Figure 136. The input data structure of an event pattern.

Figure 137 below shows the specific rule that is invoked to handle its corresponding event type.

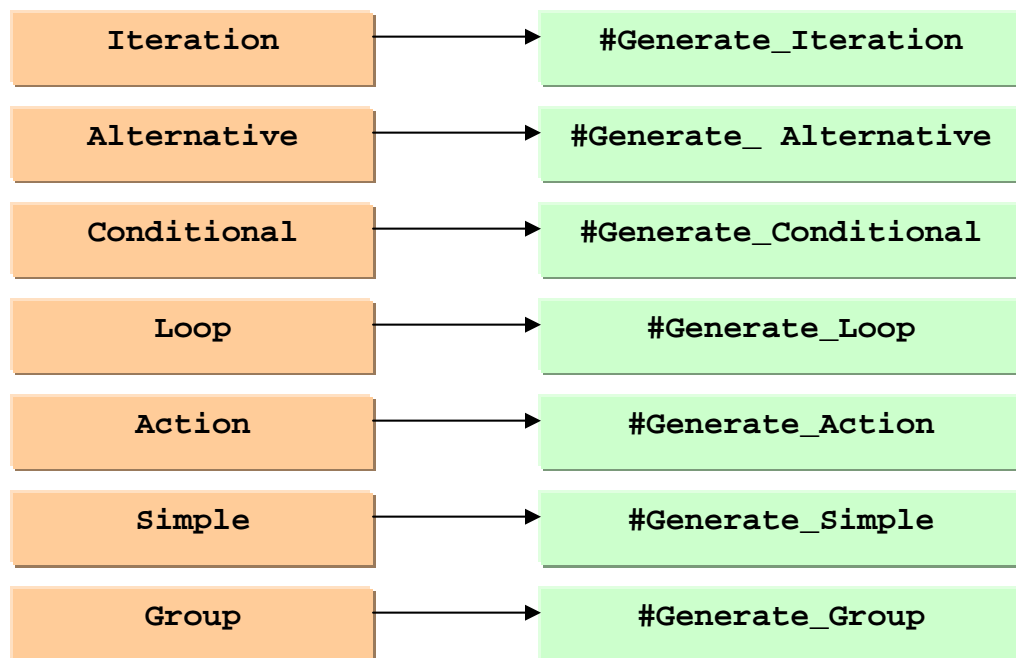


Figure 137. Mapping of code generating rules to event types.

g. *The #Generate_Iteration Rule*

This rule expects an iteration event pattern with a data structure shown in Figure 138 below:

```
<. Repeat : ( . $Op $Expression . ),  
  This: $PatternList .>  
$num_of_occurences
```

Figure 138. Input to #Generate_Iteration Rule.

There are 3 possible outputs to the input depending on the type of operator specified in \$Op. The meaning for each of these operators is summarised in Table 2 below:

Operator Expression	Meaning
= Expression	Repeat the list of event patterns EXACTLY the number of times specified by Expression
<= Expression	Repeat the list of event patterns a random number of times LESS THAN or EQUAL to the number specified by Expression.
< Expression	Repeat the list of event patterns a random number of times LESS THAN the number specified by Expression.

Table 3. Iteration Operators.

If the operator \$Op is = then the equivalent C++ code generated is shown in Figure 139. The rule #Generate_Patterns is called recursively for each pattern in the list \$PatternList. The \$num_of_occurences variable is used to differentiate between multiple instances of the same event within each rule.

```

for (__int64 i=1; i=$Expression; i++)
{
    // Specific code returned by RIGAL rule
    // #Generate_Patterns for n number of event patterns
    #Generate_Patterns($PatternList[1], $num_of_occurences)
    .
    .
    .
    #Generate_Patterns($PatternList[n], $num_of_occurences)
}

```

Figure 139. Code generated by **#Generate_Iteration** when **\$Op = '='**.

If the operator **\$Op** is **<=** or **<**, the generated C++ code will include an uniquely named integer variable to hold the value of the number of iterations to be performed. A unique number **\$Unique** that is global to the RIGAL rule **#Generate** is incremented and appended to the name of the integer **Chance** each time an iteration pattern is encountered. All global variables are accessed using the RIGAL key word **LAST** followed by the parent rule identifier that defines its scope. A number is then randomly generated between 1 and **\$Expression** and stored in the **Chance** integer which is then used in the C++ **for** loop expression. The RIGAL source code that does the above and the output C++ code are shown in Figure 140 ad 141 respectively.

```

IF ($Op = '=') ->
    GEN_CHAINS <<
    'for (__int64 i=1; i<= ' $Expression ' ; i++)';
ELSIF (T) ->
    $Unique := LAST #Generate $Unique + 1;

    GEN_CHAINS <<
    'int Chance'$Unique ' = Rand() % ' $Expression ' ;';

    GEN_CHAINS <<
    'for (__int64 i=1; i ' $Op ' Chance' $Unique ' ; i++)';
    $UniqueFI;

```

Figure 140. Portion of the RIGAL source code to implement iteration.

```

int Chance$Unique;
Chance$Unique = Rand() & $Expression;
for (__int64 i=1; i $Op Chance$Unique; i++)
{
    // Specific code returned by RIGAL rule
    // #Generate_Patterns for n number of event patterns
    #Generate_Patterns($PatternList[1], $num_of_occurences)
    .
    .
    .
    #Generate_Patterns($PatternList[n], $num_of_occurences)
}

```

Figure 141. Code generated by **#Generate_Iteration** when **\$Op** is not '='.

*h. The **#Generate_Alternative** and related rules*

The **#Generate_Alternative** rule expects an input pattern shown in Figure 142 comprising a list of alternative outcomes followed by the **\$num_of_occurences** value that is used to differentiate between multiple instances of the same alternative event within each rule.

```

$List_of_Outcomes
$num_of_occurences

```

Figure 142. Input to **#Generate_Alternative** Rule.

The rule generates a C++ statement to generate a random variable **Chance** as well as creating 3 variables **\$PlusSign**, **\$ElseSign** and **\$Cumulative** that will be used by the **\$Generate_Probability** rule.

The **\$Generate_Probability** rule is called to generate the equivalent C++ code for each of the outcomes in **\$List_of_Outcomes**. Repeated calling of this rule is necessary to create a nested series of C++ IF-ELSE statements in the generated code as illustrated in Figure 143. The RIGAL source code that does the above is shown in Figure 144.


```

if (Chance$Unique < $Prob[1])
{
    #Generate_Patterns
    ($PatternList[1] $num_of_occurence)
}
else
if (Chance$Unique < $Prob[1]+ $Prob[2])
{
    #Generate_Patterns
    ($PatternList[2] $num_of_occurence)
}
.
.
else
if (Chance$Unique < $Prob[1]+ $Prob[2] + ... + $Prob[n])
{
    #Generate_Patterns
    ($PatternList[n] $num_of_occurence)
}

```

Figure 143. Code generated by `#Generate_Probability` for n outcomes.

```

$Unique := LAST #Generate $Unique + 1;

GEN_CHAINS <<
'double Chance'$Unique ' = (double) rand() / RAND_MAX;';

$PlusSign := NULL;
$ElseSign := NULL;
$Cumulative := NULL;

FORALL $Outcome IN $List_of_Outcomes
DO
    #Generate_Probability
    ($Unique $Outcome $num_of_occurrences)
OD;

```

Figure 144. Portion of the RIGAL source code to implement alternatives.

For this current implementation of the code generator, the C++ built-in pseudorandom random variate generator, the `rand()` function is used to generate the number `Chance`. The `rand()` function generates integers in the range [0, RAND_MAX] inclusive with RAND_MAX being a value defined in the `stdlib.h` being typically 32767. The `srand()` function is often used to seed this pseudorandom variate generator prior to calling the function `rand()`.

The METALS code generator can easily be extended by substituting the the statement printing the code dealing with pseudo random variate generation shown in Figure 145 below with any standard pseudorandom variate generators used in actual combat simulation. Listings of common continuous and discrete pseudorandom variate generators [JERRY] are given in Table 3 and 4 respectively.

```
GEN_CHAINS <<  
'double Chance'$Unique ' = (double) rand() / RAND_MAX;';
```

Figure 145. Standard print the C++ pseudorandom number generator in the output.

Pseudorandom Variates	Probability Distribution Function	Pseudo Code
Continuous Uniform	$F(x) = \begin{cases} 0 & x \leq a \\ \frac{x-a}{b-a} & a < x < b \\ 1 & x \geq b \end{cases}$	Let U = Random(0,1) Return X = a + (b-a) U
Continuous Exponential	$F(x) = \begin{cases} 1 - \exp\left(-\frac{x}{a}\right) & x > 0 \\ 0 & \text{otherwise} \end{cases}$	Let U = Random(0,1) Return X = -a ln(1-U)
Continuous Weibull	$F(x) = \begin{cases} 1 - \exp\left[-\left(\frac{x}{a}\right)^b\right] & x > 0 \\ 0 & \text{otherwise} \end{cases}$	Let U = Random(0,1) Return X = -a ln(1-U) ^{1/b}
Continuous Normal	No closed form expression for distribution. Distribution mean = μ Distribution variance = σ^2	While (true) { Let U ₁ = Random(0,1) Let U ₂ = Random(0,1) Let V ₁ = 2U ₁ - 1 Let V ₂ = 2U ₂ - 1 Let W = V ₁ ² + V ₂ ² If (W < 1) { Let Y = [(-2 ln W) / W] ^{1/2} Return X ₁ = $\mu + \sigma V_1 Y$ Return X ₂ = $\mu + \sigma V_2 Y$ } }

Table 4. Continuous Pseudorandom Variate Generators.

Pseudorandom Variates	Probability Mass Function	Pseudo Code
Discrete Bernoulli	$X = \begin{cases} 1 & \text{probability} = p \\ 0 & \text{probability} = 1 - p \end{cases}$	Let U = Random(0,1) If (U ≤ p) Return X = 1 Else Return X = 0
Discrete Binomial	$P(x) = \begin{cases} \binom{n}{x} p^x (1-p)^{n-x} & x = 0, 1, \dots, n \\ 0 & \text{otherwise} \end{cases}$ <p>where $\binom{n}{x} = \frac{n!}{x!(n-x)!}$</p>	Let the function BER(n,p) implements the mass function and returns the value of X for a given n and p. X = 0 For (i = 1 to n) Let B = BER(n,p), X=X+B Return X

Discrete Poisson	$P(x) = \begin{cases} \frac{e^{-\lambda} \lambda^x}{x!} & x = 0, 1, \dots \\ 0 & \text{otherwise} \end{cases}$	Let a = exp(-λ) Let p = 1 Let X = 1 While (p > a) { Let U = Random(0,1) p = pU X = X + 1 } Return X
---------------------	--	---

Table 5. Discrete Pseudorandom Variate Generators.

The `$Generate_Probability` rule expects an input pattern shown in Figure 146.

```
$Unique
<. Patterns: $PatternList,
  Probability: $Prob .>
$num_of_occurences
```

Figure 146. Input to `#Generate_Probability` Rule.

During code generation, the `$ElseSign` is printed before each IF statement. In order to get the correct sequence nested IF-ELSE statements, this `$ElseSign` variable is initially set to NULL before being assigned with the string `ELSE` after the first IF statement.

Likewise the required cumulative probability values for successive IF statements is printed using the `$Cumulative` variable that stores the probability value `$Prob` for each outcome preceding by a `$PlusSign` variable storing the plus sign `+`. For the first IF statement, the plus sign is set to NULL to get a correct syntax.

The rule `$Generate_Patterns` is called recursively for each pattern in the list `$PatternList`. The RIGAL source code that does all the above is shown in Figure 147.

```

LAST #Generate_Alternative $Cumulative :=
LAST #Generate_Alternative $PlusSign
+ ' ' + $Prob + ' ' +
LAST #Generate_Alternative $Cumulative;

$PlusSign := LAST #Generate_Alternative $PlusSign;
$ElseSign := LAST #Generate_Alternative $ElseSign;
$Cumulated := LAST #Generate_Alternative $Cumulative;

GEN_CHAINS << $ElseSign;
GEN_CHAINS << 'if (Chance'$Unique < ' $Prob ')';
GEN_CHAINS << '{';
FORALL $Pattern IN $PatternList
DO
    #Generate_Patterns($Pattern $num_of_occurences);
OD;
GEN_CHAINS << '}';

```

Figure 147. Portion of the RIGAL source code to implement IF-ELSE statements.

i. The #Generate_Conditional Rule

The **#Generate_Conditional** rule expects an input pattern shown in Figure 148 comprising a Boolean expression **\$Bool**, two alternative event patterns followed by the **\$num_of_occurences** value that is used to differentiate between multiple instances of the same conditional event within each rule.

```

<. When: $Bool,
    Do: $Pattern, [Else: $Pattern2] .>
$num of occurences

```

Figure 148. Input to **#Generate_Alternative** Rule.

The rule generates an equivalent IF-ELSE C++ statement shown in Figure 149. The rule **\$Generate_Patterns** is called recursively for the two alternative patterns.

```

if ($Bool)
{
    #Generate_Patterns($Pattern $num_of_occurences)
}
else
{
    #Generate_Patterns($Pattern2 $num_of_occurences)
}

```

Figure 149. Code generated by `#Generate_Conditional`.

j. *The #Generate_Loop Rule*

The `#Generate_Loop` rule expects an input pattern shown in Figure 150 comprising a Boolean expression `$Bool`, an event pattern followed by the `$num_of_occurences` value that is used to differentiate between multiple instances of the same loop event within each rule.

```

<. While : $Bool,
    Do : $Pattern .>
$num of occurences

```

Figure 150. Input to `#Generate_Loop` Rule.

The rule generates an equivalent WHILE C++ statement shown in Figure 151. The rule `$Generate_Patterns` is called recursively for the pattern.

```

while ($Bool)
{
    #Generate_Patterns($Pattern $num_of_occurences)
}

```

Figure 151. Code generated by `#Generate_Loop`.

k. The #Generate_Action Rule

The `#Generate_Action` rule expects an input stream of string tokens that represents C++ code. These tokens are output directly by the rule to the source file without modifications.

l. The #Generate_Simple Rule

The `#Generate_Simple` rule expects an input pattern shown in Figure 152 comprising an event name, its order of occurrence within the rule value `$Order` followed by the `$num_of_occurences` value that is used to determine if a previous instance of the same event already exist within the rule.

```
<. Event_Name: $Id,  
order: $Order .>  
$num_of_occurences
```

Figure 152. Input to `#Generate_Simple` Rule.

The rule generates an equivalent C++ functional call to the event function with the name `$Id` shown in Figure 153. If previous instances of the same event already exists within the rule (`$num_of_occurences > 1`), than an array index in the form of `[$Order]` is added immediately after the name of event `$Id[$Order]`.

```
$Id[$Order] = $Id_rule ( $Id[$Order] );
```

Figure 153. Code generated directly by `#Generate_Simple`.

The event function always return an event object. This is to allow attributes for the current instance of an event to be accessible and modifiable by other events. This important feature forms the basis of how computations can be done over an event trace (series of events within a rule).

m. The #Generate_Group Rule

The #Generate_Group rule expects an input pattern shown in Figure 154 comprising a list (group) of event patterns \$Group_Of_Patterns followed by the \$num_of_occurences value that is used to differentiate between multiple instances of the same group event within each rule.



```
$Group_Of_Patterns
$num_of_occurences
```

Figure 154. Input to #Generate_Alternative Rule.

The rule \$Generate_Patterns is then called recursively for each of the event pattern in \$Group_Of_Patterns.

The complete METALS lexical analyzer, parser and code generator source code in RIGAL is given in Appendix I.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. APPLICATION OF METALS

A. EXAMPLE 1 - SIMPLE COIN TOSS.

A simple coin example is first used to demonstrate the basic capabilities of METALS. In this example, a coin is flipped a million times and the total number of times that the result is a 'head' is counted. The probabilities for obtaining a 'head' or 'tail' are both set at 0.5. The input source code written in METALS is shown in Figure 155 below:

```
SIMULATE CoinFlip { Version 1 }

ENTITY Simulation
{ BEGIN __int64 TotalCoinFlips; __int64 NumHeads; END }

EVENT Initialize
EVENT Head
EVENT Tail
EVENT Start_Flipping

CHAIN Algo
{
Start_Flipping : Initialize
                REPEAT(=3000000000)
                { BEGIN Simulation.TotalCoinFlips++; END
                  DECIDE ( P(0.5) Head | P(0.5) Tail ) }
                BEGIN
                  cout << "Total Coin Flips = "
                    << Simulation.TotalCoinFlips << "\n"
                    << "Number of Heads = "
                    << Simulation.NumHeads << "\n";
                END;

Initialize : BEGIN Simulation.TotalCoinFlips = 0;
              Simulation.NumHeads = 0; END;

Head : BEGIN Simulation.NumHeads++; END;

Tail;
}
```

Figure 155. METALS source code for Simple Coin Toss example.

The simulation named **CoinFlip** comprises a single entity which is the **Simulation** itself and 4 events, namely **Initialize**, **Head**, **Tail** and **Start_Flipping**.

The Simulation entity contains 2 attributes **TotalCoinFlips** and **NumHeads** which are modified by a single chain of events named **Algo**. The simulation start point is the first rule in the first chain, which in this simple example is the rule **Start_Flipping**. **Start_Flipping** begins with the event **Initialize**, followed by an **Iteration** event pattern and an **Action** event pattern.

The Iteration event pattern itself contains an **Action** pattern followed by an **Alternative** event pattern which can call either the **Head** or **Tail** event. Specific code for the **Initialize**, **Head** and **Tail** events finally completes the event chain.

The intermediate parsed tree for the METALS program produced by the METALS parser is given in Figure 156. This tree is then fed to the METAL code generator to generate an equivalent C++ program comprising of the following 5 output files:

- MAIN.CPP - This is the main C++ program file.
- WORLDS.CPP - This is the C++ header file containing class definitions for simulation environments. For this example, no world is defined in the source code so the header file is empty.
- ENTITIES.CPP - This is the C++ header file containing class definitions for simulation entities.
- EVENTS.CPP - This is the C++ header file containing event class definitions.
- CHAINS.CPP - This is the C++ header file that containing functions that implements event rules.

Listings for these output files are given in Figures 157 to 160.

```
<.Title: <.Title: 'CoinFlip',
  Description: (. 'Version' '0' '1'B' .).>,

Entities: (.<.Entity_Name: 'Simulation',
  Entity_Attributes: (. '__int64' 'TotalCoinFlips' ';'
    '__int64' 'NumHeads' ';' .).>.),

Events: (.<.Event_Name: 'Initialize'.>
  <.Event_Name: 'Head'.>
  <.Event_Name: 'Tail'.>
  <.Event_Name: 'Start_Flipping'.>.),

Chains: (.<.Chain_Name: 'Algo',
  Rules: (.

<.Rule_Name: 'Start_Flipping',

  Patterns: (.<.Type: 'Simple',
    Body: <.Event_Name: 'Initialize',
    order: 0.>.>

    <.Type: 'Iteration',
    Body: <.Repeat: (. '=' (.3000000000.) .),
      This: (.<.Type: 'Action',
        Body: (. 'Simulation' '.' 'TotalCoinFlips' '++' ';' .).>

        <.Type: 'Alternative',
        Body: (.<.Patterns: (.<.Type: 'Simple',
          Body: <.Event_Name: 'Head',
          order: 0.>.>.),
          Probability: '0.5'.>

          <.Patterns: (.<.Type: 'Simple',
            Body: <.Event_Name: 'Tail',
            order: 0.>.>.),
            Probability: '0.5'.>.>.>.>

        <.Type: 'Action',
        Body: (. 'cout' '<<' 'Total Coin Flips = ' '<<' 'Simulation' '.'
          'TotalCoinFlips' '<<' '\n' ';' 'cout' '<<' 'Number of Heads = '
          '<<' 'Simulation' '.' 'NumHeads' '<<' '\n' ';' .).>.),

      Used_Event_Names: <.Initialize: 'T',
        Head: 'T',
        Tail: 'T'.>,

      number_of_occurrences: <.Initialize: 1,
        Head: 1,
        Tail: 1.>.>

    <.Rule_Name: 'Initialize',

    Patterns: (.<.Type: 'Action',
      Body: (. 'Simulation' '.' 'TotalCoinFlips' '=' '0' ';'
        'Simulation' '.' 'NumHeads' '=' '0' ';' .).>.>

    <.Rule_Name: 'Head',

    Patterns: (.<.Type: 'Action',
      Body: (. 'Simulation' '.' 'NumHeads' '++' ';' .).>.>

    <.Rule_Name: 'Tail'.>.>.>.>

  )>.)>.
```

Figure 156. Intermediate parsed tree produced by METALS parser.

```

// METALS Code Generator Version 0.4C
// C++ simulation program [CoinFlip] created from aaa.txt
// Version 0 . 1 B

#include <iostream>
#include <ctime>
#include <cstdlib>
#include <stdlib.h>

using namespace std;

#include "Worlds.h"
#include "Entities.h"
#include "Events.h"
#include "Chains.h"

// Main
int main(int argc, char *argv[])
{
    Start_Flipping_class Start_Flipping;
    Start_Flipping_rule( Start_Flipping );

    system("PAUSE");
    return 0;
}

```

Figure 157. Listings for MAIN.CPP generated by METALS code generator.

```

// METALS Code Generator Version 0.4C
// Generation of Entities

class Simulation_class
{
public:
    _int64 TotalCoinFlips;
    _int64 NumHeads;
};

Simulation_class Simulation;

// Base class for all events
long Global_Event_Count;

enum EType { DET, STO };

class Event
{
public:
    Event(EType, double);
    string Name;
    EType EventType;
    double Mean;
    double Duration;
};

Event :: Event(EType EventType, double Mean)
{
    Global_Event_Count++;

    if (EventType==DET) Duration = Mean;
    else { Duration = Mean; }
}

```

Figure 158. Listings for ENTITIES.H generated by METALS code generator.

```

// METALS Code Generator Version 0.4C
// Generation of Event Classes

long Global_Event_Count;

enum EType { DET, STO };

class Initialize_class
{
public:
    Initialize_class();
    string Name;
    EType Var;
    double Mean;
    double Duration;
    static long Instances;
};

long Initialize_class :: Instances = 0;

Initialize_class :: Initialize_class()
{
    Global_Event_Count++;
    Instances++;

    Name = "Initialize";
    Var = DET;
    Mean = 0;
    Duration = 0;

    if (Var==STO) // Discrete Poisson RNG.
    {
        double a = exp(-Mean);
        double p = 1;
        long x = 1;
        while (p > a) {
            double U = (double) rand() / RAND_MAX;
            p = p*U;
            x++; }
        Duration = x; }
}

```

Figure 159. Partial Listings for EVENTS.H generated by METALS code generator.

```

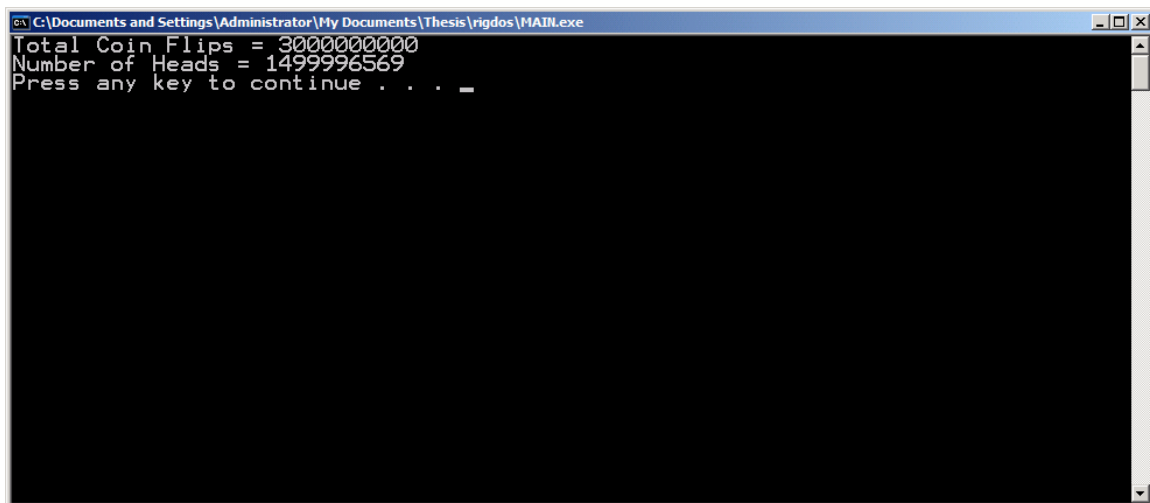
// METALS Code Generator Version 0.4C
// Generation of Event Chains
// The various user defined event descriptions in Algo
// Functional prototypes for each rule in the event chain
Start_Flipping_class Start_Flipping_rule( Start_Flipping_class );
Initialize_class Initialize_rule( Initialize_class );
Head_class Head_rule( Head_class );
Tail_class Tail_rule( Tail_class );

Start_Flipping_class Start_Flipping_rule( Start_Flipping_class )
{
    Start_Flipping_class Start_Flipping;
    Initialize_class Initialize;
    Head_class Head;
    Tail_class Tail;
    Initialize =Initialize_rule (Initialize);
    for (__int64 i=1; i<=1000000; i++)
    {
        Simulation . TotalCoinFlips ++;
        double Chance = (double) rand() / RAND_MAX;
        if (Chance < 0.5 )
        {
            Head =Head_rule (Head);
        }
        else
        if (Chance < 0.5 )
        {
            Tail =Tail_rule (Tail);
        }
    }
    cout << "Total Coin Flips = " << Simulation . TotalCoinFlips << "\n";
    cout << "Number of Heads = " << Simulation . NumHeads << "\n";

    return Start_Flipping;
}

```

Figure 160. Listings for CHAINS.H generated by METALS code generator.



```

C:\Documents and Settings\Administrator\My Documents\Thesis\rigdos\MAIN.exe
Total Coin Flips = 3000000000
Number of Heads = 1499996569
Press any key to continue . . . _

```

Figure 161 Screen output by program after execution.

In this simulation, a coin is flipped 3 billion times and the total number of heads obtained is counted. The C++ `__int64` data type is used to support a maximum of 9,223,372,036,854,775,808 runs using its 8 bytes or 64 bit size. The entire simulation took a total of 552 seconds on a Pentium 4 2.4 GHz notebook with 512 Mb of RAM. This gives an estimated 5.4 million runs per second for this simple experiment.

B. EXAMPLE 2 - MINE AVOIDANCE IN LITTORALS

1. Introduction

The use of minefields at sea is the oldest but nevertheless still high effective and widely used naval tactic due to its simplicity in execution and the low cost of mines.

Minefields are normally deployed at navigational chokepoints to impede or delay the safe passage of ships. This makes littoral waters, with its intrinsic characteristics of shallow waters, cluttered seabed and constricted passages ideal environments for the deployment of mines. Mines are cheap relative to the sophisticated modern navy vessels, and therefore constitutes a significant weapon of asymmetry against navies.

There are two widely debated approaches that modern navies can use to create a safe passage through a minefield. The first is the classical method of Mine-Clearance, whereby, a mine clearance vessel will detect, classify and destroy mines along a pre-determined path through the affected area. This method is extremely time consuming, because not only will a vessel need time to classify and identify an object detected on sonar, time is needed to physically position charges near a mine to destroy it upon positive identification. The expected higher population density of non-mine, mine-like bottom objects (NOMBOs) in littoral waters can potentially make the method of mine-clearance untenable in terms of how long the entire operation is going to take.

The second method is a more radical method of Mine-Avoidance, whereby, a mine hunting vessel will detect, classify, mark NOMBOs using special mine avoidance sonars and changing course to avoid them while continuing to go through a mine field. This is analogous to the classic maze problem whereby a 'mouse' is supposed to work through a labyrinth or walls and dead ends to reach the exit on the other side of the maze.

The effectiveness of both methods can be measured in terms of:

- Total time taken to create a safe navigational path of a certain size through the affected area.
- The probability of successful safe transit of the mine clearance/hunting vessel during its operation. In other words, whether the vessel indeed survive while conducting mine clearance/avoidance operations.

The typical state space diagrams for both mine clearance and avoidance operations are shown in Figure 162 and 163 respectively.

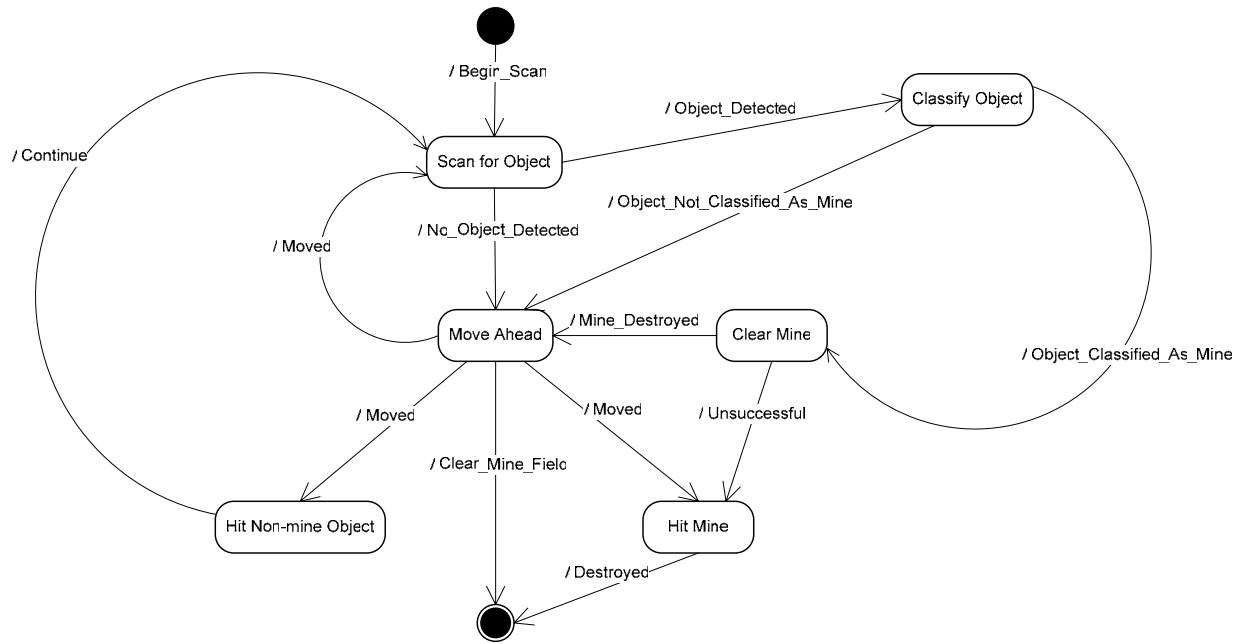


Figure 162. Mine Clearance State Diagram.

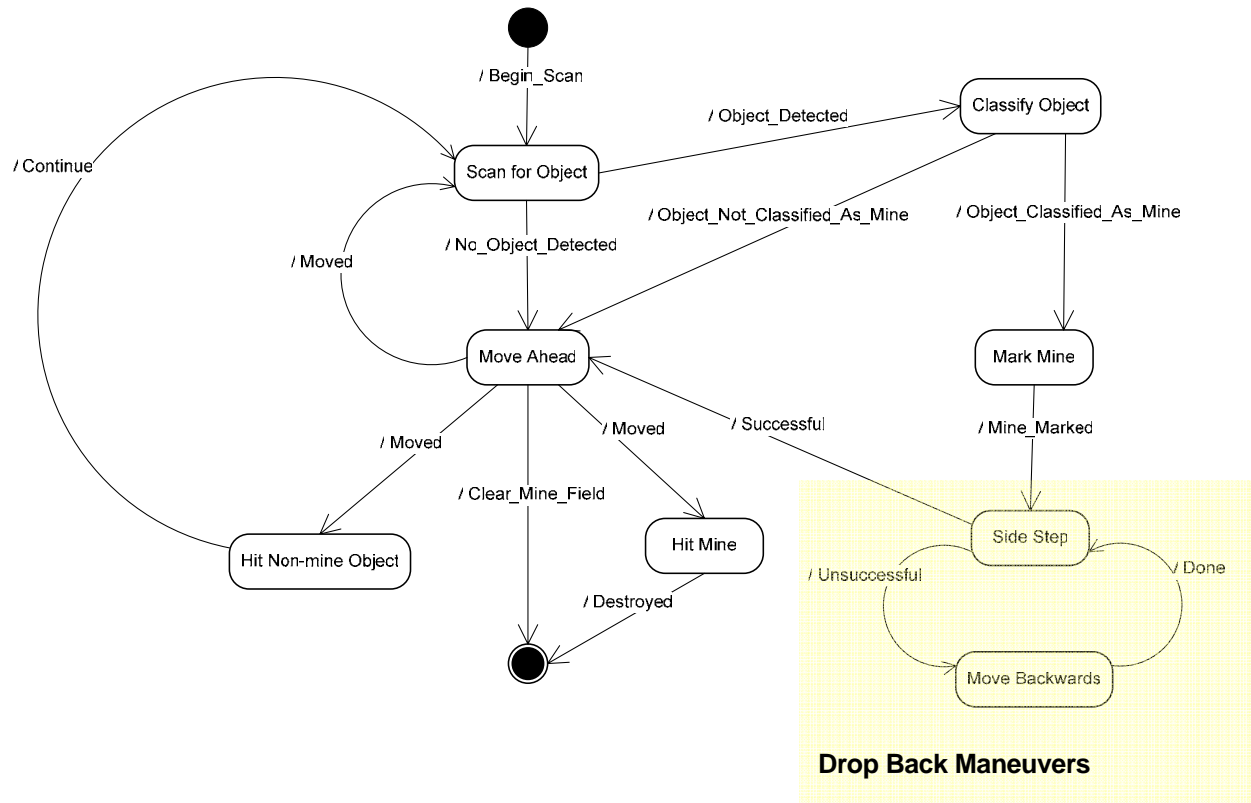


Figure 163. Mine Avoidance State Diagram.

It is beyond the scope of this thesis to analyze in depth the effectiveness of both methods using extensive simulations. Instead, a Mine Avoidance simulation program with its complicated search and decision making routines will be built from ground up based on the state diagram shown above using METALS exclusively as a demonstration of the its capabilities.

2. Event Space During Mine Avoidance Operations

The distinction between events in the design domain and events in the problem domains had been made earlier in Chapter 2. While such a distinction is important in the design of the METALS language, for the end-user, the METALS language does not differentiate between the two types of events and supports all event definitions with the same language constructs. For the Mine Avoidance problem, the list of possible events that will take place during simulation is summarized in Table 5 below.

Event ID	Description	Domain
Initialize	All global variables are initialized during this event. All worlds and entities are instantiated from their respective class definitions.	Design
Populate	The simulation environment in this case a minefield is populated randomly with a range of user defined objects.	Design
Start_Simulation	Simulation begins with this event which encompass a number of events representing simulation runs.	Design
Start_Run	Each Start_Run event encapsulates all problem domain events.	Design
Move	Vessel begins to scan ahead for objects. If object is detected and classified as NOMBOS then avoid by calling either the Go_Clockwise event or Go_AntiClockwise event, else proceed ahead and repeat the move event again.	Problem
Go_Clockwise	Circumscribe detected mine-like objects clockwise.	Problem
Go_AntiClockwise	Circumscribe detected mine-like objects clockwise.	Problem

Table 6. Mine Avoidance Events.

The event trace for this simulation program is shown in Figure 164 below:

```
Simulation_Program: Initialize Populate Start_Simulation
Start_Simulation   : Start_Run
Start_Run          : Move*
Move               : Go_Clockwise | Go_AntiClockwise
```

Figure 164. Event Trace for Mine Avoidance.

3. Defining The Simulation Entities

For this simulation, the list of entities created is listed in Table 6 below, and the corresponding METALS source code to create them is shown in Figure 165.

Entity	Attributes	Description
Simulation	MaxRuns = Maximum number of runs to execute during the simulation	The overall simulation program.
	numClockwise = Performance indicator counting the number of clockwise movements	
	numAntiwise = Performance indicator counting the number of anticlockwise movements	
Run	MaxSteps = Maximum number of steps to iterate through to prevent infinite looping	Each simulation run.
Ship	Clockwise = Boolean value indicating whether current mode of circumscribing is clockwise or otherwise.	The mine-avoiding vessel.
	CurrentCol, CurrentRow = Current position within the minefield.	
	StartCol, FinishCol = Starting and finishing position.	

Table 7. Mine Avoidance Entities.

```

ENTITY Simulation { BEGIN
    long MaxRuns, numClockwise, numAntiClockwise;
END }
ENTITY Run { BEGIN long MaxSteps; END }
ENTITY Ship { BEGIN
    bool Clockwise;
    int CurrentCol, CurrentRow, StartCol, FinishCol;
END }

```

Figure 165. Entities needed in the simulation.

4. The Mine Avoidance Algorithm

With reference to the event trace given in Figure 165, the mine avoidance algorithm is encapsulated within the event **Move** in the simulation. The basic pseudo code for the **Move** algorithm implemented in C++ is shown in Figure 166 below:

```
Scan one square North.
If no object detected {
    Move one square North, Mark current square as BeenHere }
else {
    Scan one square North-East.
    If no object detected {
        Move one square North-East, Mark current square as BeenHere }
    else {
        Scan one square North-West
        If no object detected {
            Move one square North-West, Mark current square as BeenHere }
        else {
            If Mode is clockwise {
                Circumscribe clockwise around objects }
            else {
                Circumscribe anticlockwise around objects }
        }
    }
}
```

Figure 166. Pseudo-code for Event Move.

The circumscription routines around objects are based on the wall following maze solving routine with some customization for this simulation. The **Go_Clockwise** and **Go_AntiClockwise** events are called to move the vessel by one step clockwise or anticlockwise around objects ahead of the vessel. Upon reaching a new position, the **Move** event is again called and the sequence of events thus repeats..

5. Modeling The Minefield.

The METALS **World** language construct is used here to create the 2-D array representing the minefield. A minefield can contain a few standard types of objects such as a NOMBO, a non-mine object, a sonar false alarm or a mine. The **TERRAINS** keyword in the **World** grammar is used here to define a range of possible objects or event markers that can be associated with each element of the array. The METALS source code for establishing and setting up a Minefield is shown in Figure 167 below:

```
WORLD Minefield {  
    ROWS = 32  
    COLS = 32  
    TERRAINS = ( Empty Nombo NonMine False_Alarm Mine  
                Boundary Start Finish )  
    ATTRIBS = ( BEGIN // Additional Attribs Can Be Added Here END )  
}
```

Figure 167. METALS source code to set up a minefield.

After creating a minefield with the required attributes, the next step is to populate the minefield with objects whose types have just been defined. An event **Populate** is defined for this purpose. Inside this event, the expected number of mines and NOMBOs inside the Minefield are computed based on expected mine and object population density. The METALS source code based using the Action pattern (C++ Code) for this computation is shown in Figure 168 below:

```
int Objects_Per_Square_km= 5;  
int Mines_Per_Square_km = 3;  
int Size_Of_Square = 200;  
int Num_Squares_Per_Square_km =  
(1000/Size_Of_Square)*(1000/Size_Of_Square);  
  
int ExpectedNumMines = (NumSquares / Num_Squares_Per_Square_km ) *  
    Mines_Per_Square_km;  
int ExpectedNumObjects = (NumSquares / Num_Squares_Per_Square_km ) *  
    Objects_Per_Square_km;
```

Figure 168. METALS source code calculate the expected number of mines and NOMBOs.

Continuing with the **Populate** event, using the expected values for the number of mines and NOMBOs, and using the total number of squares in the 2-D array Minefield, mines and NOMBOs are first distributed randomly across the total number of squares using a uniform randomly number generation. The code that spread mines and objects over the total number of squares is shown in Figure 169 below:

```
int i, j, Choice;
srand((unsigned)time(0));
for (i=0; i<=ExpectedNumObjects-1; i++) {
    Choice = 1 + rand() % NumSquares;
    IsObject[Choice-1] = true;
}
for (j=0; j<=ExpectedNumMines-1; j++) {
    Choice = 1 + rand() % NumSquares;
    if (IsObject[Choice-1] == false)
        IsMine[Choice-1] = true;
    else {
        IsObject[Choice-1] == false;
        IsNombo[Choice-1] = true;
    }
}
```

Figure 169. Algorithm for spreading mines and objects.

From the code shown in Figure 169, 3 arrays **IsObject**, **IsMine** and **IsNombo**, each with an array size that is equal to the total number of squares are used store information about whether any particular square contains a mine, NOMBO or object. These array are then used to set the object type for each element in the Minefield shown in Figure 170 below:

```
int CurrentPointer = 0;
for (int Row=1; Row<=Minefield_MaxRows-2; Row++) {
    for (int Col=1; Col<=Minefield_MaxCols-2; Col++) {
        Minefield[Row][Col].ObjectType = Empty;
        if (IsMine[CurrentPointer] == true)
            Minefield[Row][Col].ObjectType = Mine;
        if (IsObject[CurrentPointer] == true)
            Minefield[Row][Col].ObjectType = NonMine;
        if (IsNombo[CurrentPointer] == true)
            Minefield[Row][Col].ObjectType = Nombo;
        CurrentPointer++;
    }
}
```

Figure 170. Algorithm for setting the object type for each Minefield square.

It is to be noted that the distribution of objects (NOMBOs) and mines within the 2-D array is typically simulation based on the Spatial Poisson Process that is extremely well documented and presented in [GAVER1] and [GAVER2]. A summary of the Spatial Poisson Process from these documents are summarized in Figure 171 below.

Spatial Poisson Process

- Assumptions
 - Number of objects in disjoint increments of area are independent
 - One object in each small increment of area
- Poisson rate λ is the average number of objects per unit area
- Random number of objects in sub-region of area a , $N(a)$, is Poisson distributed
- Amounts of area, A , covered between objects is exponentially distributed
- $$P\{N(a) = n\} = e^{-\lambda a} \frac{[\lambda a]^n}{n!}$$
- Subscripts used for Poisson rates, i.e., average numbers of mines and NOMBOs, λ_M and λ_O , respectively
- $$P\{A > a\} = P\{N(a) = 0\} = e^{-\lambda a} \frac{[\lambda a]^0}{0!} = e^{-\lambda a}$$
- Mines & NOMBOs are independently placed

Figure 171. Spatial Poisson Process for Mine Avoidance.

6. Computations Over Event Traces.

In this simulation, besides finding a clear path through a minefield, the total and average duration per simulation run, as well as the total number of clockwise and anticlockwise movements are also parameters of interests. The METALS source code for computing the total duration for the total number of simulation runs is shown in Figure 172 below. The **Start_Run** event is executed **MaxRuns** number of times and the duration for each **Start_Run** is accumulated in the attribute **Duration** for the entity **Simulation**. Simple calculations are next done to obtain the average duration per run. The same approach is used to keep track of the number of clockwise and anticlockwise movements.

```

REPEAT(=Simulation.MaxRuns) {
    Start_Run
    BEGIN
        Start_Simulation.Duration += Start_Run.Duration;
    END
}

BEGIN
    double Avg_Duration = Start_Simulation.Duration / Simulation.MaxRuns;
    cout << "Average duration per run over " << Simulation.MaxRuns
        << " run(s) is " << Avg_Duration << "\n";
    cout << "Total number of events occurred = " << Global_Event_Count
        << "\n";
    cout << "Number of Clockwise Movements = "
        << Simulation.numClockwise << "\n";
    cout << "Number of AntiClockwise Movements = "
        << Simulation.numAntiClockwise << "\n";
END;

```

Figure 172. Computing duration over event traces.

METALS is design to allow event attributes to be exposed to other events by having each event rule return a copy of the same event type object. This allow computations like summation and averaging to be perform in a straightforward, tidy and highly controlled manner.

7. **Compiling The METALS Source And Building The Simulation Program.**

The complete METALS source code for the Mine Avoidance Simulation is given in Appendix II. It is This 263 lines in length METALS source code is fed into the METALS parser and code generator to produce a C++ simulation program that comprises 6 output files with a total of 682 lines. This gives a target to source length ratio of about 2.6.

As METALS is designed for computations over event traces, it is leaves the bulk of attribute manipulation to the C++ language in the forms of actions. Despite this, the savings in terms of the number of lines in absolute terms as well ss the higher level of abstraction offered by METALS is significant.

8. Simulation Results.

A simulation over 1000000 runs is conducted with the ship starting at the same position. The minefield is repopulated after even run. The compiled C++ simulation program is executed 3 times, each running a total of 1000000 simulation runs. The results for all three executions are given in Table 8 below:

	1st Execution 1	2nd Execution 2	3rd Execution	Average
Number of Simulation Runs Conducted	1 000 000	1 000 000	1 000 000	1 000 000
Total Real Time Taken (s)	150	172	154	158.666
Number of Simulation Runs per Second	6666	5813	6493	6302
Average Simulation Time (Duration) per Run	120.237	122.979	123.473	122.230
Length of event trace	132 430 713	132 434 961	133 525 651	132 797 108
Total number of clockwise movements	24 095 580	25 089 616	24 727 338	24 637 511
Total number of anti-clockwise movements	15 983 512	15 903 520	16 430 360	16 105 797
Clockwise to Anticlockwise Ratio	1.507527	1.577614	1.504978	1.529729

Table 8. Results of Simulation Runs.

The mean time taken for each execution is 158.666 seconds on a Pentium 4 2.4 GHz notebook with 512 Mb of RAM. This gives an estimated 6302 runs per second performance for this simple experiment. The mean length of event trace is 132797108 or about 132 million events long. A mean total of 24637511 clockwise movements were recorded versus 16105797 anticlockwise movements, which yields a mean ratio of 1.5 to 1. This implies that the algorithm is slightly biased to favor the clockwise circumscription around objects. The screen captures from on 3 runs is shown in Figures 173 to 175 below:

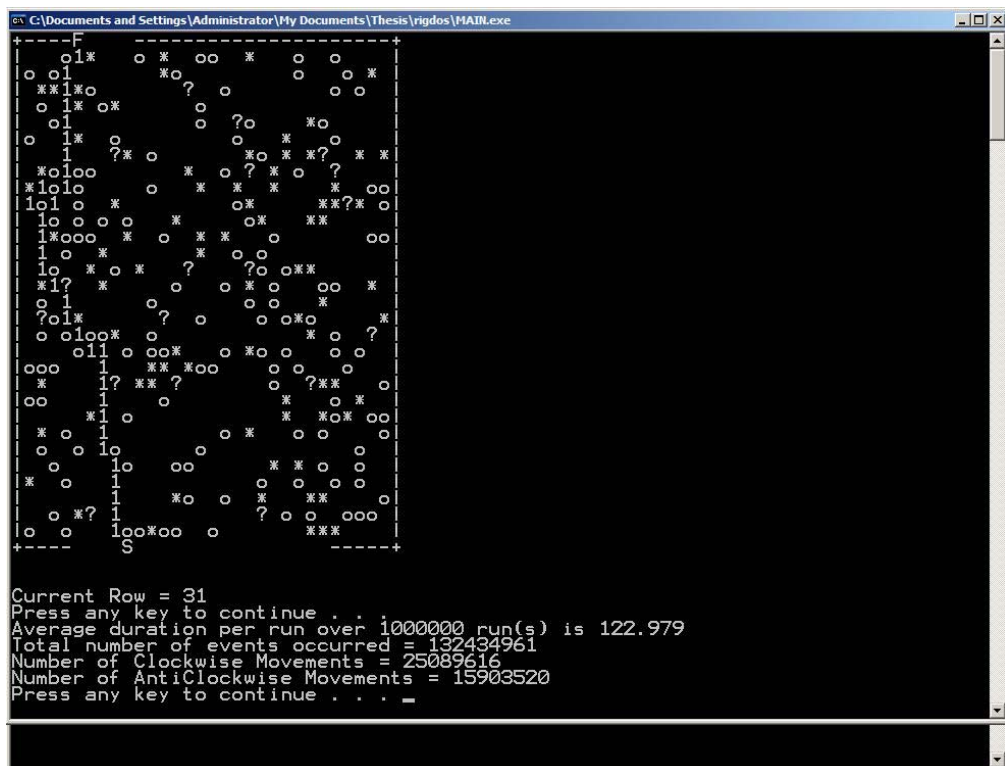




Figure 175. Screen output by Mine Avoidance simulation program after execution 3.

On the screens, the symbol o represents objects, * represents mines and ? represent NOMBOs. The numeric letters 1, 2 ... represents the number of times a particular square has been visited by the ship. The ship is transiting from bottom to top. The particular map of the minefield is the output for the final run in the simulation.

On the whole, in combat simulation, the ability to perform similar computations over event traces can proof to be significantly useful. Examples include resource management and monitoring resource depletion, such as the utilization of the port torpedo tube versus the starboard torpedo tube using a particular tactic, the amount of ammunition consumed by guns at various positions using a particular maneuver.

V. CONCLUSION

This thesis has introduced an alternative way to design and construct combat simulation models adapting methodologies like program behavior modeling, event grammars and computations over event traces that are used traditionally in software debugging automation tools.

The feasibility and benefits of the new approach has been verified first by the successful design and development of METALS and demonstrated subsequently by the building of a mine avoidance simulation program. The METALS language however is far from being complete in terms of providing the complete range of functionalities needed for real world combat simulations. Possible future work needed will include extensions and modifications to the METALS language construct to support more simulation features, most notably constructs that automate computations over event traces and perform some attribute manipulation. A high level combat model written in METALS can potentially be compiled into C++, Java or any other code by pre-built compilers by just designing the appropriate code generator, and make the model more portable and platform/environment independent. Therefore, a Java version of the METALS code generator will represent an immediate next step. In the longer run, a graphical development environment to model, create and run simulations visually is also possible.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: SOURCE CODE FOR THE METALS COMPILER

A. THE METALS LEXICAL ANALYZER 0.5

```
-- Meta Language for Combat Simulation - Compiler
#MAIN
    PRINT 'METALS Parser Version 0.4B';

    -- TOKENIZATION
    -- Obtain code written in METALS from Combat.txt and generate
list of tokens.
    $Filename := 'MINE4.txt';
    $Tokens := #CALL_PAS( 35 $Filename 'L+A-U-P-C+p-m+');
    PRINT $Tokens;

    -- LANGUAGE PARSING
    -- Use list of tokens and parse them into intermediate form.
    $Results := #Parse($Tokens);

    -- CODE GENERATION
    -- Use intermediate form to generate C++ code.
    #Generate($Results $Filename);
##
    #Generate_Patterns($Pattern $num_of_occurences);
OD;
GEN_CHAINS << '}' ;
/
##
```


B. THE METALS PARSER VERSION 0.5

```
-- Meta Language for Combat Simulation - Parser Version 0.5

-- Obtain entire parsed tree for simulation program
#Parse
    / PRINT 'Inside #Simulation_Program'; /

    (
    $Title := #Title
    [( * $Headers !.:= #Header * )]
    [( * $Worlds !.:= #World * )]
    [( * $Entities !.:= #Entity * )]
    [( * $Events !.:= #Event * )]
    [( * $Chains !.:= #Chain * )]
    .)

    / RETURN
    <. Title: $Title,
        Headers: $Headers,
        Worlds: $Worlds,
        Entities: $Entities,
        Events: $Events,
        Chains: $Chains .> /

##

-- Obtain simulation title and description
#Title
    / PRINT 'Inside #Title'; /

    'SIMULATE' $Id ['{' [$Description := #Plain_Text] '}]'
    / RETURN
    <. Title: $Id,
        Description: $Description .> /

##

-- Add the user's own header file(s)
#Header
    / PRINT 'Inside #Header'; /

    'INCLUDE' $Id
    / RETURN $Id /

##

-- Obtain user-defined world
#World
    / PRINT 'Inside #World'; /

    'WORLD' $World_Name
    '{'
    'ROWS' '=' $MaxRows
    'COLS' '=' $MaxCols
    'TERRAINS' '=' '(' (* $Terrains !.:= S'($$<> ')') *) ')'
    ['ATTRIBS' '=' '(' $Attributes := #CPP_Code ')']
```

```

    '}'
  / RETURN
  <. World_Name: $World_Name,
    MaxRows: $MaxRows,
    MaxCols: $MaxCols,
    Terrains: $Terrains,
    Attributes: $Attributes .> /
##

-- Obtain user-defined entity
#Entity
  / PRINT 'Inside #Entity'; /

  'ENTITY' $Entity_Name
  ['{'
    [$Entity_Attributes := #CPP_Code]
  '}]
  / RETURN
  <. Entity_Name: $Entity_Name,
    Entity_Attributes: $Entity_Attributes .> /
##

-- Obtain user defined events
#Event
  / PRINT 'Inside #Event'; /

  'EVENT' $Event_Name
  ['{'
    [$Event_Attributes := #Event_Attributes]
  '}]

  / RETURN
  <. Event_Name: $Event_Name,
    Event_Attributes: $Event_Attributes .> /
##

-- Used by #Event to obtain event attributes.
#Event_Attributes
  / PRINT 'Inside #Event_Attributes'; /

  [$Var := ('STO'!'DET')]
  ['MEAN' '=' $Mean]
  ['DUR' '=' $Duration]
  [$Additional_Attribs := #CPP_Code]
  /
  -- Assign default values.
  IF ($Var = NULL) -> $Var := 'DET'; FI;
  IF ($Mean = NULL) -> $Mean := '0'; FI;
  IF ($Duration = NULL) -> $Duration := '0'; FI;
  IF ($Additional_Attribs = NULL) -> $Additional_Attribs :=
'None'; FI;
  RETURN
  <. Var: $Var,
    Mean: $Mean,
    Duration: $Duration,
    Additional_Attribs: $Additional_Attribs .> /
##

```

```

-- Obtain user defined chains of events
#Chain
    / PRINT 'Inside #Chain'; /

    'CHAIN' $Chain_Name
    '{'
        (* $Rules !.:= #Rule *)
    '}'
    / RETURN
    <. Chain_Name: $Chain_Name,
        Rules: $Rules .> /
##

-- Used by #Chain to obtain individual sequences or rules.
#Rule
    / PRINT 'Inside #Rule'; /

-- global var: $used_event_names : <* $id: T *>
-- $num_of_occurrences: <* $Id: $Numb *>
-- number of rule name $Id occurrences in the right hand part

    $Rule_Name [ ':' (* $Patterns !.:= #Pattern *) ] ';'
    / RETURN
    <. Rule_Name: $Rule_Name,
        Patterns: $Patterns,
        Used_Event_Names: $used_event_names,
        number_of_occurrences: $num_of_occurrences .> /

--
;;
-- (* $List !.:= S'($$ <> ';' ) *)
-- / PRINT 'Syntax error in #Rule found in the following:';
-- PRINT $List; /
##

-- Standard patterns for various data and control structures.
#Pattern
    / PRINT 'Inside #Pattern'; /

    -- Initialize pattern type
    / $Type := NULL; /

    -- Match pattern
    $Pattern := ( #Iteration      / $Type := 'Iteration';      / !
                  #Loop           / $Type := 'Loop';           / !
                  #Conditional    / $Type := 'Conditional';    / !
                  #Alternative     / $Type := 'Alternative';    / !
                  #Action         / $Type := 'Action';         / !
                  #Group          / $Type := 'Group';          / !
                  #Simple         / $Type := 'Simple';         / )
    / RETURN <. Type: $Type, Body: $Pattern .> /
    ;;
    '(' (* $List !.:= #Pattern *) ')'
    / RETURN $List /
##

```

```

#Iteration
    / PRINT 'Inside RIGAL Rule Iteration'; /
    'REPEAT' '(' $Op := #Operator $Expression := #Plain_Text2 ')' '{'
(* $Pattern !.:= #Pattern *) '}'
    / RETURN <. Repeat : (. $Op $Expression .), This: $Pattern .> /
    --> ONFAIL PRINT 'Iteration Pattern Matching Failure'
##

#Loop
    / PRINT 'Inside #Loop'; /
    'WHILE' '(' $Bool := #Bool_Expression ')' '{' (* $Pattern !.:=
#Pattern *) '}'
    / RETURN <. While : $Bool, Do : $Pattern .> /
    --> ONFAIL PRINT 'Conditional Pattern Matching Failure'
##

#Conditional
    'WHEN' '(' $Bool := #Bool_Expression ')' '{' (* $Pattern !.:=
#Pattern *) '}'
    [ 'ELSE' '{' (* $Pattern2 !.:= #Pattern *) '}' ]
    / RETURN <. When: $Bool, Do: $Pattern, Else: $Pattern2 .> /
    --> ONFAIL PRINT 'Conditional Pattern Matching Failure'
##

-- Pattern for decision making control structure
-- Legal patterns are as follows:
--     DECIDE ( Event1 | Event2 )                      - 2 events
--     DECIDE ( Event1 | Event2 | Event3 ... | EventN ) - N events
--     DECIDE ( P(Value1) Event1 | P(Value2) Event2 )
--     DECIDE ( P(Value1) Event1 | P(Value2) Event2 | .... | P(ValueN)
EventN )
--     Value can be '0.5' or '.5'
#Alternative
    / PRINT 'Inside #Alternative'; /

    'DECIDE'
    '('
    (* $Outcome := #Outcome / $List_Of_Outcomes !.:= $Outcome; / *
'|')
    ')'
    / RETURN $List_Of_Outcomes /
##

-- Used by #Alternative to recognize an outcome
#Outcome
    / PRINT 'Inside #Outcome'; /

    'P' '(' $Value := #Probability ')' (* $Patterns !.:= #Pattern *)
    / RETURN <. Patterns: $Patterns, Probability: $Value .> /
    ;;
    (* $Patterns !.:= #Pattern *) / $Value := 'unknown'; /
    / RETURN <. Patterns: $Patterns, Probability: $Value .> /
##

-- Used by #Alternative to obtain probability value between 0 and 1
#Probability

```

```

    / PRINT 'Inside #Probability'; /

    (* $List !.:= S'($$ <> ' )' ) *)
    / IF ($List[1] = '.') -> $List := (. 0 .)!!$List; FI;
    $Value := #IMPLode($List);
    RETURN $Value /
##

#Simple
    $Id

    / LAST #Rule $used_event_names ++:= <. $Id:T .>;
    LAST #Rule $num_of_occurrences ++:= <. $Id: LAST #Rule
$num_of_occurrences.$Id +1 .>;

    RETURN
    <. Event_Name: $Id,
    order: LAST #Rule $num_of_occurrences.$Id - 1 .> /
##

#Group
    '(' (* $Group_Of_Patterns !.:= #Pattern *) ')'
    / RETURN $Group_Of_Patterns /
##

-- Pattern for user defined C++ code
#Action
    / PRINT 'Inside #Action'; /

    $Action := #CPP_Code
    / RETURN $Action /
##

-- Obtain boolean expression from list of tokens
#Bool_Expression
    / PRINT 'Inside #Bool_Expression'; /

    $Bool_Expression := #CPP_Code
    / RETURN $Bool_Expression /
##

-- Used by #Iteration to recognize and obtain C++ operator from
token(s)
#Operator
    / PRINT 'Inside #Operator'; /
    (
        ( '<' '=' / $Operator := '<=' / ) !
        ( '<'      / $Operator := '<' / ) !
        ( '='      / $Operator := '=' / )
    )
    / RETURN $Operator /
##

-- Obtain plain text from list of tokens
#Plain_Text
    / PRINT 'Inside #Plain_Text'; /

```

```

    (* (
      ( '/' '/' / $Plain_Text !.:= '//' /) !
      ( '<' '=' / $Plain_Text !.:= '<=' /) !
      ( '>' '=' / $Plain_Text !.:= '>=' /) !
      ( '=' '=' / $Plain_Text !.:= '==' /) !
      ( '!' '=' / $Plain_Text !.:= '!= ' /) !
      ( '+' '+' / $Plain_Text !.:= '++' /) !
      ( '-' '-' / $Plain_Text !.:= '--' /) !
      ( '+' '=' / $Plain_Text !.:= '+=' /) !
      ( '-' '=' / $Plain_Text !.:= '-=' /) !
      ( '|' '|' / $Plain_Text !.:= '||' /) !
      ( '&' '&' / $Plain_Text !.:= '&&' /) !
      ( '<' '<' / $Plain_Text !.:= '<<' /) !
      ( '>' '>' / $Plain_Text !.:= '>>' /) !
      $Plain_Text !.:= S'($$<> '}'')
    ) *)
  / RETURN $Plain_Text /
##

-- Obtain plain text from list of tokens
#Plain_Text2
  / PRINT 'Inside #Plain_Text'; /

  (* (
    ( '/' '/' / $Plain_Text !.:= '//' /) !
    ( '<' '=' / $Plain_Text !.:= '<=' /) !
    ( '>' '=' / $Plain_Text !.:= '>=' /) !
    ( '=' '=' / $Plain_Text !.:= '==' /) !
    ( '!' '=' / $Plain_Text !.:= '!= ' /) !
    ( '+' '+' / $Plain_Text !.:= '++' /) !
    ( '-' '-' / $Plain_Text !.:= '--' /) !
    ( '+' '=' / $Plain_Text !.:= '+=' /) !
    ( '-' '=' / $Plain_Text !.:= '-=' /) !
    ( '|' '|' / $Plain_Text !.:= '||' /) !
    ( '&' '&' / $Plain_Text !.:= '&&' /) !
    ( '<' '<' / $Plain_Text !.:= '<<' /) !
    ( '>' '>' / $Plain_Text !.:= '>>' /) !
    $Plain_Text !.:= S'($$<> '}'')
  ) *)
  / RETURN $Plain_Text /
##

-- Obtain C++ code from list of tokens
#CPP_Code
  / PRINT 'Inside #CPP_Code'; /
  'BEGIN'
  (* (
    ( '/' '/' / $CPP_Code !.:= '//' /) !
    ( '<' '=' / $CPP_Code !.:= '<=' /) !
    ( '>' '=' / $CPP_Code !.:= '>=' /) !
    ( '=' '=' / $CPP_Code !.:= '==' /) !
    ( '!' '=' / $CPP_Code !.:= '!= ' /) !
    ( '+' '+' / $CPP_Code !.:= '++' /) !
    ( '-' '-' / $CPP_Code !.:= '--' /) !
    ( '+' '=' / $CPP_Code !.:= '+=' /) !
    ( '-' '=' / $CPP_Code !.:= '-=' /) !
    ( '|' '|' / $CPP_Code !.:= '||' /) !

```

```

( '&' '&' / $CPP_Code !.:= '&&' /) !
( '<' '<' / $CPP_Code !.:= '<<' /) !
( '>' '>' / $CPP_Code !.:= '>>' /) !
$CPP_Code !.:= S'($$<> 'END')
) *)
'END'
/ RETURN $CPP_Code /
##

```

C. THE METALS CODE GENERATOR 0.5

```
-- Meta Language for Combat Simulation - Code Generator Version 0.5

-- Generate equivalent C++ simulation program from intermediate
representation
#Generate
    / PRINT 'Inside #Generate'; /

    <. Title: $Title,
      [Headers: $Headers],
      [Worlds: $Worlds],
      [Entities: $Entities],
      [Events: $Events],
      [Chains: $Chains] .>
$Filename

/
$Unique := 1;
$First_Rule := NULL;

-- Open CPP files to add code
OPEN GEN_MAIN 'Main.cpp';
OPEN GEN_WORLD 'Worlds.h';
OPEN GEN_ENTITIES 'Entities.h';
OPEN GEN_EVENTS 'Events.h';
OPEN GEN_CHAINS 'Chains.h';

-- Main Program Headers

#Generate_Title($Title $Filename );
#Generate_Headers($Headers);
#Generate_Worlds($Worlds);
#Generate_Entities($Entities);
#Generate_EventClasses($Events);
#Generate_EventChains($Chains);

-- Main Program
GEN_MAIN <<;
GEN_MAIN << '// Main';
GEN_MAIN << 'int main(int argc, char *argv[])';
GEN_MAIN << '{';
GEN_MAIN << '    '@ $First_Rule.Rule_Name '_class '
$First_Rule.Rule_Name ' ';
GEN_MAIN << '    '@ $First_Rule.Rule_Name '_rule' @ '( '
$First_Rule.Rule_Name ' )';
GEN_MAIN <<;
GEN_MAIN << '        system("PAUSE");';
GEN_MAIN << '        return 0;';
GEN_MAIN << '    }';
GEN_MAIN <<;
/
##
```



```

#Generate_Title
    / PRINT 'Inside #Generate_Title'; /

    <. Title: $Title,
      Description: $Description .>
    $Filename

    /
    GEN_MAIN << '// METALS Code Generator Version 0.4C';
    GEN_MAIN << @ '// C++ simulation program [' $Title '] created
from ' $Filename;
    GEN_MAIN << '// ';
    FORALL $Atom IN $Description
    DO
        GEN_MAIN <] @ $Atom ' ';
    OD;
    /
##

#Generate_Headers
    / PRINT 'Inside #Generate_Headers'; /

    $Headers

    /
    GEN_MAIN <<;
    GEN_MAIN << '#include <iostream>';
    GEN_MAIN << '#include <ctime>';
    GEN_MAIN << '#include <cstdlib>';
    GEN_MAIN << '#include <stdlib.h>';
    GEN_MAIN <<;
    GEN_MAIN << 'using namespace std;';
    GEN_MAIN <<;
    GEN_MAIN << '#include "Worlds.h"';
    GEN_MAIN << '#include "Entities.h"';
    FORALL $HEADER IN $Headers
    DO
        GEN_MAIN << @ '#include "' $HEADER '.h"';
    OD;
    GEN_MAIN << '#include "Events.h"';
    GEN_MAIN << '#include "Chains.h"';
    GEN_MAIN <<;

    /
##

#Generate_Worlds
    / PRINT 'Inside #Generate_Worlds'; /

    $Worlds

    /
    $Comma := NULL;

```

```

$Tail := NULL;

GEN_WORLD << '// METALS Code Generator Version 0.4C';
GEN_WORLD <<;
GEN_WORLD << '// Generation of Worlds';
GEN_WORLD <<;

FORALL $World IN $Worlds
DO
    GEN_WORLD << @ 'int ' $World.World_Name '_MaxRows = '
$World.MaxRows ';;
    GEN_WORLD << @ 'int ' $World.World_Name '_MaxCols = '
$World.MaxCols ';;
    GEN_WORLD <<;
    GEN_WORLD << @ 'enum ' $World.World_Name '_Terrain {';
    FORALL $Terrain IN $World.Terrains
    DO
        IF $Terrain = $World.Terrains[-1] -> $Tail := ' }';
FI;

    GEN_WORLD <] @ $Comma ' ' $Terrain $Tail;
    IF $Comma = NULL -> $Comma := ','; FI;

OD;
GEN_WORLD <<;
$Comma := NULL;
$Tail := NULL;
GEN_WORLD << @ 'class ' $World.World_Name '_class';
GEN_WORLD << '{';
GEN_WORLD << '    public:;
GEN_WORLD << @ '        ' $World.World_Name '_class();';
GEN_WORLD << '        int BeenHere;';
GEN_WORLD << '        bool Marked;';
GEN_WORLD << @ '        ' $World.World_Name '_Terrain
ObjectType;';
GEN_WORLD << '    ';
FORALL $Attribute IN $World.Attributes
DO
    IF $Attribute = ';' -> GEN_WORLD <] @ $Attribute;
GEN_WORLD << '    ';
    ELSIF $Attribute <> ';' -> GEN_WORLD <] @ ' '
$Attribute; FI;
OD;
GEN_WORLD << '};';
GEN_WORLD <<;
GEN_WORLD << @ $World.World_Name '_class :: '
$World.World_Name '_class();';
GEN_WORLD << '{';
GEN_WORLD <<;
GEN_WORLD << '    BeenHere = 0;';
GEN_WORLD << '    Marked = false;';
GEN_WORLD << '    ObjectType = Empty;';
GEN_WORLD << '};';
GEN_WORLD <<;
GEN_WORLD << @ $World.World_Name '_class '
$World.World_Name '[' $World.MaxRows '[' $World.MaxCols '];';
GEN_WORLD <<;

OD;
/

```

```

##

#Generate_Entities
/ PRINT 'Inside #Generate_Entities'; /

$Entities

/
GEN_ENTITIES << '// METALS Code Generator Version 0.4C';
GEN_ENTITIES <<;
GEN_ENTITIES << '// Generation of Entities';
GEN_ENTITIES <<;

-- Define C++ classes for every user defined entity
FORALL $Entity IN $Entities
DO
    GEN_ENTITIES << @ 'class ' $Entity.Entity_Name '_class';
    GEN_ENTITIES << '{';
    GEN_ENTITIES << '    public:';
    -- GEN_ENTITIES << @ '    ' $Entity.Entity_Name
'_class()'; -- Constructor Option.
    GEN_ENTITIES <<;
    GEN_ENTITIES << '    ';
    FORALL $Attribute IN $Entity.Entity_Attributes
    DO
        IF $Attribute = ';' -> GEN_ENTITIES <] @ $Attribute;
GEN_ENTITIES << '    ';
        ELSIF $Attribute <> ';' -> GEN_ENTITIES <] @ ' '
$Attribute; FI;
    OD;
    GEN_ENTITIES << '};';
    GEN_ENTITIES <<;
OD;

-- Instantiate global entities
FORALL $Entity IN $Entities
DO
    GEN_ENTITIES << @ $Entity.Entity_Name '_class '
$Entity.Entity_Name ' ';
OD;
GEN_ENTITIES <<;
/
##

#Generate_EventClasses
/ PRINT 'Inside #Generate_EventClasses.'; /

$Events

/
GEN_EVENTS << '// METALS Code Generator Version 0.4C';
GEN_EVENTS <<;
GEN_EVENTS << '// Generation of Event Classes';
GEN_EVENTS <<;
GEN_EVENTS << 'long Global_Event_Count;';
GEN_EVENTS <<;
GEN_EVENTS << 'enum EType { DET, STO };';

```

```

GEN_EVENTS <<;
-- Define C++ classes for every user defined event
FORALL $Event IN $Events
DO
    GEN_EVENTS << @ 'class ' $Event.Event_Name '_class';
    GEN_EVENTS << '{';
    GEN_EVENTS << '    public:';
    GEN_EVENTS << @ '        ' $Event.Event_Name '_class()';
    GEN_EVENTS << '        string Name;';
    GEN_EVENTS << '        EType Var;';
    GEN_EVENTS << '        double Mean;';
    GEN_EVENTS << '        double Duration;';
    GEN_EVENTS << '        static long Instances;';
    GEN_EVENTS << '    ';
    IF $Event.Event_Attributes.Additional_Attribs <> 'None' ->
        FORALL $Attribute IN
$Event.Event_Attributes.Additional_Attribs
        DO
            IF $Attribute = ';' -> GEN_EVENTS <] @
$Attribute; GEN_EVENTS << '    ';
            ELSIF T -> GEN_EVENTS <] @ '    ' $Attribute; FI;
        OD;
    FI;
    GEN_EVENTS <<;
    GEN_EVENTS << '};';
    GEN_EVENTS <<;
    GEN_EVENTS << @ 'long ' $Event.Event_Name '_class ::
Instances = 0;';
    GEN_EVENTS <<;
    GEN_EVENTS << @ $Event.Event_Name '_class :: '
$Event.Event_Name '_class()';
    GEN_EVENTS << '{';
    GEN_EVENTS << '    Global_Event_Count++;';
    GEN_EVENTS << '    Instances++;';
    GEN_EVENTS <<;

    GEN_EVENTS << @ '        Name = " ' $Event.Event_Name ' "';

    IF $Event.Event_Attributes.Var = NULL -> GEN_EVENTS << '
Var = DET;';
    ELSIF T -> GEN_EVENTS << '        Var = '
$Event.Event_Attributes.Var ';; FI;

    IF $Event.Event_Attributes.Mean = NULL -> GEN_EVENTS << '
Mean = 0;';
    ELSIF T -> GEN_EVENTS << '        Mean = '
$Event.Event_Attributes.Mean ';; FI;

    IF $Event.Event_Attributes.Duration = NULL -> GEN_EVENTS <<
'        Duration = 0;';
    ELSIF T -> GEN_EVENTS << '        Duration = '
$Event.Event_Attributes.Duration ';; FI;

    GEN_EVENTS <<;
    GEN_EVENTS << '        if (Var==STO) // Discrete Poisson RNG.';
    GEN_EVENTS << '        {';

```

```

GEN_EVENTS << '          double a = exp(-Mean);';
GEN_EVENTS << '          double p = 1;';
GEN_EVENTS << '          long x = 1;';
GEN_EVENTS << '          while (p > a) {';
GEN_EVENTS << '              double U = (double) rand() /
RAND_MAX;';
GEN_EVENTS << '              p = p*U;';
GEN_EVENTS << '              x++; }';
GEN_EVENTS << '          Duration = x; }';
GEN_EVENTS << '      }';
GEN_EVENTS <<;

OD;
/
##

#Generate_EventChains
/ PRINT 'Inside #Generate_EventChains.'; /

$Chains

/
GEN_CHAINS << '// METALS Code Generator Version 0.4C';
GEN_CHAINS <<;
GEN_CHAINS << '// Generation of Event Chains';
GEN_CHAINS <<;

-- Generate C++ code for each event chain
FORALL $Chain IN $Chains
DO
    GEN_CHAINS << @ '// The various user defined event
descriptions in '$Chain.Chain_Name';
    GEN_CHAINS <<;

    -- Set the starting point for the simulation.
    LAST #Generate $First_Rule := $Chain.Rules[1];

    -- Generate C++ functional prototypes for each rule in the
event chain.
    GEN_CHAINS << '// Functional prototypes for each rule in
the event chain';

    FORALL $Rule IN $Chain.Rules
    DO
        IF LAST #Generate $First_Rule = NULL -> LAST
#Generate $First_Rule := $Rule; FI;
        GEN_CHAINS << @ $Rule.Rule_Name '_class '
$Rule.Rule_Name '_rule( ' $Rule.Rule_Name '_class );';
        OD;
        GEN_CHAINS <<;

        -- Generate C++ code for each rule.
        FORALL $Rule IN $Chain.Rules
        DO
            GEN_CHAINS <<;
            #Generate_Rules($Rule);
            $Set.($Rule.Rule_Name) := generated;

```

```

        OD;
        GEN_CHAINS <<;
        FORALL $r IN $set
        DO
            IF $set.$r = T -> GEN_CHAINS << @ $r '_class ' $r
'_Rule( ' $r '_class ){}';' FI
        OD;
    OD;
/
##

#Generate_Rules
/ PRINT 'Inside #Generate_Rules.'; /

<. Rule_Name: $Rule_Name,
[Patterns: $Patterns],
[Used_Event_Names: $used_event_names],
[number_of_occurrences: $num_of_occurrences] .>

/
GEN_CHAINS << @ $Rule_Name '_class ' $Rule_Name '_rule( '
$Rule_Name '_class )';
GEN_CHAINS << '{';
GEN_CHAINS << @ ' ' $Rule_Name '_class ' $Rule_Name ' ';
GEN_CHAINS <<;

-- Instantiate event classes used in this rule.
FORALL $EVAR IN $used_event_names
DO
    GEN_CHAINS << @ $EVAR '_class ' $EVAR;
    IF $num_of_occurrences.$EVAR > 1 -> GEN_CHAINS <] '['
$num_of_occurrences.$EVAR ']' FI;
    GEN_CHAINS <]';
OD;

-- Generate rest of code for this rule.
FORALL $Pattern IN $Patterns
DO
    #Generate_Patterns($Pattern $num_of_occurrences);
OD;
GEN_CHAINS <<;
GEN_CHAINS << @ ' ' return ' $Rule_Name ' ';
GEN_CHAINS << '>';
GEN_CHAINS <<;
/
##

#Generate_Patterns
/ PRINT 'Inside #Generate_Patterns.'; /

<. Type: $Type, Body: $Pattern .>
$num_of_occurrences

/
IF $Type='Iteration' -> #Generate_Iteration($Pattern
$num_of_occurrences);

```

```

        ELSIF $Type='Alternative'      -> #Generate_Alternative($Pattern
$num_of_occurences);
        ELSIF $Type='Conditional'      -> #Generate_Conditional($Pattern
$num_of_occurences);
        ELSIF $Type='Loop'             -> #Generate_Loop($Pattern
$num_of_occurences);
        ELSIF $Type='Action'           -> #Generate_Action($Pattern
$num_of_occurences);
        ELSIF $Type='Simple'           -> #Generate_Simple($Pattern
$num_of_occurences);
        ELSIF $Type='Group'            -> #Generate_Group($Pattern
$num_of_occurences) FI;
    /
    ;;
    (. (* #Generate_Patterns *) .)
    --;;
    --/
    -- GEN_CHAINS << '// Placeholder for rule without pattern(s).'
    --/
##

#Generate_Simple
    / PRINT 'Inside #Generate_Simple.'; /

    <. Event_Name: $Id, order: $Order .>
    $num_of_occurences

    /
    GEN_CHAINS << '      ' $Id;

    IF $num_of_occurences.$Id > 1
        -> GEN_CHAINS <] @[' '$Order ']'
    FI;

    GEN_CHAINS <] @ '=' $Id '_rule (' $Id;

    IF $num_of_occurences.$Id > 1 -> GEN_CHAINS <] @[' '$Order ']'
FI;

    GEN_CHAINS <] ');';
    /
##

#Generate_Loop
    / PRINT 'Inside #Generate_Loop.'; /

    <. While : $Bool,
        Do : $Pattern .>
    $num_of_occurences

    /
    GEN_CHAINS << @ '      while (' $Bool ')';
    GEN_CHAINS << @ '{';
    #Generate_Patterns($Pattern $num_of_occurences);
    GEN_CHAINS << @ '}';
    /
##

```

```

#Generate_Group
  / PRINT 'Inside #Generate_Group.'; /

  $Group_Of_Patterns
  $num_of_occurences

  /
  FORALL $Pattern IN $Group_Of_Patterns
  DO
    #Generate_Patterns($Pattern $num_of_occurences);
  OD;
  /
##

#Generate_Action
  / PRINT 'Inside #Generate_Action.'; /

  $Action
  $num_of_occurences

  /
  FORALL $Atom IN $Action
  DO
    IF $Atom = ';' -> GEN_CHAINS <] @ $Atom; GEN_CHAINS << '
';
    ELSIF #TATOM($Atom) -> GEN_CHAINS <] @ #CHR(34)
#IMPLode($Atom) #CHR(34); GEN_CHAINS << '      ';
    ELSIF $Atom <> ';' -> GEN_CHAINS <] @ ' ' $Atom; FI;
  OD;
  /
##

#Generate_Conditional
  / PRINT 'Inside #Generate_Conditional.'; /

  <. When: $Bool, Do: $Pattern, [Else: $Pattern2] .>
  $num_of_occurences

  /
  GEN_CHAINS << @ '      if ( ' $Bool ')';
  GEN_CHAINS << @ '{';
  #Generate_Patterns($Pattern $num_of_occurences);
  GEN_CHAINS << @ '}';
  GEN_CHAINS << @ 'else';
  GEN_CHAINS << @ '{';
  #Generate_Patterns($Pattern2 $num_of_occurences);
  GEN_CHAINS << @ '}';
  /
##

#Generate_Alternative
  / PRINT 'Inside #Generate_Alternative.'; /

  -- (. <. Pattern: <. Type:'Simple', Body: <.
Event_Name:'Head',order:0 .> .>, Probability:'0.5' .>

```



```

--      <. Pattern: <. Type:'Simple', Body: <.
Event_Name:'Tail',order:0 .> .>, Probability:'0.5' .> .)
  $List
  $num_of_occurrences

/
GEN_CHAINS << 'double Chance = (double) rand() / RAND_MAX;';
$PlusSign := NULL;
$ElseSign := NULL;
$Cumulative := NULL;

FORALL $Outcome IN $List
DO
    #Generate_Probability($Outcome $num_of_occurrences) OD;
/
##

#Generate_Probability
/ PRINT 'Inside #Generate_Probability.'; /

-- <. Patterns: (. <. Type: 'Simple', Body: <. Event_Name:
'Head', order:0.> .> .), Probability:'0.5' .>
<. Patterns: $PatternList, Probability: $Prob .>
$num_of_occurrences

/
LAST #Generate_Alternative $Cumulative := LAST
#Generate_Alternative $PlusSign + ' ' + $Prob + ' ' + LAST
#Generate_Alternative $Cumulative;
$PlusSign := LAST #Generate_Alternative $PlusSign;
$ElseSign := LAST #Generate_Alternative $ElseSign;
$Cumulated := LAST #Generate_Alternative $Cumulative;

GEN_CHAINS << $ElseSign;
GEN_CHAINS << 'if (Chance < ' $Prob ')';
GEN_CHAINS << '{';
FORALL $Pattern IN $PatternList
DO
    #Generate_Patterns($Pattern $num_of_occurrences);
OD;
GEN_CHAINS << '}';

LAST #Generate_Alternative $PlusSign := '+';
LAST #Generate_Alternative $ElseSign := 'else';
/
##

#Generate_Iteration
/ PRINT 'Inside #Generate_Iteration.'; /

<. Repeat : (. $Op $Expression .), This: $PatternList .>
$num_of_occurrences

/
IF ($Op='=') -> GEN_CHAINS << @ 'for (__int64 i=1; i<='
$Expression '; i++)';
ELSIF (T) ->

```

```

        $Unique := LAST #Generate $Unique + 1;
        GEN_CHAINS << @ '__int64 Chance' $Unique ' = Rand() % '
$Expression ' ';
        GEN_CHAINS << @ 'for (__int64 i=1; i' $Op 'Chance' $Unique
'; i++)';
    FI;

    GEN_CHAINS << '{';
    FORALL $Pattern IN $PatternList
    DO
        #Generate_Patterns($Pattern $num_of_occurences);
    OD;
    GEN_CHAINS << '}';
/
##

```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B : MINE AVOIDANCE SIMULATION IN METALS

A. THE MINE AVOIDANCE SIMULATION METALS SOURCE CODE

```
SIMULATE Mine_Avoidance { Version 0.5A }

INCLUDE Settings

WORLD Minefield {
    ROWS = 32
    COLS = 32
    TERRAINS = ( Empty Nombo NonMine False_Alarm Mine Explosion Boundary Start
Finish )
    ATTRIBS = ( BEGIN // Additional Attribs Can Be Added Here END )
}

ENTITY Simulation { BEGIN long MaxRuns, numClockwise, numAntiClockwise; END}
ENTITY Run { BEGIN long MaxSteps; END }
ENTITY Ship { BEGIN bool Clockwise; int CurrentCol, CurrentRow, StartCol, FinishCol;
END }

EVENT Initialize
EVENT Move
EVENT Go_Clockwise { DET MEAN = 6 DUR = 6 }
EVENT Go_AntiClockwise { DET MEAN = 6 DUR = 6 }
EVENT Start_Simulation
EVENT Start_Run
EVENT Populate

CHAIN Algorithm1 {
    Start_Simulation: BEGIN
        Simulation.MaxRuns = 100000;
        Simulation.numClockwise = 0;
        Simulation.numAntiClockwise = 0;
    END
    REPEAT(=Simulation.MaxRuns) {
        Start_Run
        BEGIN
            Start_Simulation.Duration += Start_Run.Duration;
        END
    }
    BEGIN
        Draw();
    }
```

```

        double Avg_Duration = Start_Simulation.Duration / Simulation.MaxRuns;
        cout << "Average duration per run over " << Simulation.MaxRuns
            << " run(s) is " << Avg_Duration << "\n";
        cout << "Total number of events occurred = " << Global_Event_Count <<
        "\n";

        cout << "Number of Clockwise Movements = " <<
        Simulation.numClockwise << "\n";
        cout << "Number of AntiClockwise Movements = " <<
        Simulation.numAntiClockwise << "\n";
        END;

Start_Run: Initialize
    Populate
    BEGIN
        long Steps=0;
        while ((Ship.CurrentRow <= Minefield_MaxRows-
2)&&(Steps<Run.MaxSteps)) {
            Move_class Move;
            Move = Move_rule ( Move );
            Start_Run.Duration += Move.Duration;
            // Draw();
            Steps++;
        }
        // Draw();
    END;

Initialize: BEGIN
    Run.MaxSteps = 200;
    NumSquares = (Minefield_MaxCols-2) * (Minefield_MaxRows-2);

    for (int Row=0; Row<=Minefield_MaxRows-1; Row++)
        for (int Col=0; Col<=Minefield_MaxCols-1; Col++) {
            Minefield[Row][Col].Marked = false;
            Minefield[Row][Col].BeenHere = 0;
            Minefield[Row][Col].ObjectType = Empty;
        }
    int Entry_Gap[2] = { 5, 25 };
    int Exit_Gap[2] = { 5, 9 };

    for (int Col=0; Col <=Minefield_MaxCols-1; Col++) {
        if ((Col<Entry_Gap[0])||(Col>Entry_Gap[1]))
            Minefield[0][Col].ObjectType = Boundary;
        else
            Minefield[0][Col].ObjectType = Empty;

        if ((Col<Exit_Gap[0])||(Col>Exit_Gap[1]))

```

```

        Minefield[Minefield_MaxRows-1][Col].ObjectType = Boundary;
    else
        Minefield[Minefield_MaxRows-1][Col].ObjectType = Empty;

    if ((Col==0)|| (Col==Minefield_MaxCols-1)) {
        for (int Row=1; Row <=Minefield_MaxRows-2; Row++)
            Minefield[Row][Col].ObjectType = Boundary;
        }
    }

    Ship.Clockwise = true;

    srand((unsigned)time(0));
    Ship.CurrentCol = rand() % (Entry_Gap[1]-Entry_Gap[0]);
    // Ship.CurrentCol = 15;
    Ship.CurrentRow = 0;
    Ship.StartCol = Ship.CurrentCol;
END;

```

```

Populate: BEGIN
    bool IsObject[NumSquares], IsMine[NumSquares], IsNombo[NumSquares];

    int Objects_Per_Square_km= 5;
    int Mines_Per_Square_km = 3;
    int Size_Of_Square = 200;
    int Num_Squares_Per_Square_km =
(1000/Size_Of_Square)*(1000/Size_Of_Square);

    int ExpectedNumMines = (NumSquares / Num_Squares_Per_Square_km ) *
Mines_Per_Square_km;
    int ExpectedNumObjects = (NumSquares / Num_Squares_Per_Square_km ) *
Objects_Per_Square_km;

    int x;
    for (x=0; x<=NumSquares-1; x++) {
        IsMine[x] = false;
        IsObject[x] = false;
        IsNombo[x] = false;
    }

    int i, j, Choice;
    srand((unsigned)time(0));
    for (i=0; i<=ExpectedNumObjects-1; i++) {
        Choice = 1 + rand() % NumSquares;
        IsObject[Choice-1] = true;
    }

```

```

for (j=0; j<=ExpectedNumMines-1; j++) {
    Choice = 1 + rand() % NumSquares;
    if (IsObject[Choice-1] == false)
        IsMine[Choice-1] = true;
    else {
        IsObject[Choice-1] == false;
        IsNombo[Choice-1] = true;
    }
}

int CurrentPointer = 0;
for (int Row=1; Row<=Minefield_MaxRows-2; Row++) {
    for (int Col=1; Col<=Minefield_MaxCols-2; Col++) {
        Minefield[Row][Col].ObjectType = Empty;
        if (IsMine[CurrentPointer] == true) Minefield[Row][Col].ObjectType = Mine;
        if (IsObject[CurrentPointer] == true) Minefield[Row][Col].ObjectType =
NonMine;
        if (IsNombo[CurrentPointer] == true) Minefield[Row][Col].ObjectType =
Nombo;
        CurrentPointer++;
    }
}
END;

Move: BEGIN
Minefield[Ship.CurrentRow][Ship.CurrentCol].BeenHere++;
if (Ship.CurrentRow<=Minefield_MaxRows-1) {
    if ((Minefield[Ship.CurrentRow+1][Ship.CurrentCol].ObjectType == Empty) &&
        (Minefield[Ship.CurrentRow+1][Ship.CurrentCol].BeenHere == 0))
        Ship.CurrentRow++;
    else {
        if (Ship.CurrentCol == 1) Ship.Clockwise = false;
        if (Ship.CurrentCol == Minefield_MaxCols-2) Ship.Clockwise = true;
        if (Ship.Clockwise) { Go_Clockwise_class Go_Clockwise;
            Go_Clockwise = Go_Clockwise_rule(Go_Clockwise);
            Simulation.numClockwise = Go_Clockwise.Instances;
            Move.Duration = Go_Clockwise.Duration; }
        else { Go_AntiClockwise_class Go_AntiClockwise;
            Go_AntiClockwise = Go_AntiClockwise_rule(Go_AntiClockwise);
            Simulation.numAntiClockwise = Go_AntiClockwise.Instances;
            Move.Duration = Go_AntiClockwise.Duration; }
    }
}
Ship.FinishCol = Ship.CurrentCol;
END;

```

```

Go_Clockwise: BEGIN
    if (Scan(TopLeft)==false) {
        if ((Minefield[Ship.CurrentRow+1][Ship.CurrentCol-1].BeenHere > 1) &&
            (Minefield[Ship.CurrentRow-1][Ship.CurrentCol].ObjectType == Empty))
            Ship.CurrentRow--;
        else
            { Ship.CurrentRow++; Ship.CurrentCol--; }
    }
    else {
        if (Scan(Left)==false) {
            if ((Minefield[Ship.CurrentRow][Ship.CurrentCol-1].BeenHere > 1) &&
                (Minefield[Ship.CurrentRow-1][Ship.CurrentCol].ObjectType ==
Empty))
                Ship.CurrentRow--;
            else
                Ship.CurrentCol--;
        }
        else {
            if (Scan(BottomLeft)==false) {
                if ((Minefield[Ship.CurrentRow][Ship.CurrentCol-1].BeenHere > 1) &&
                    (Minefield[Ship.CurrentRow-1][Ship.CurrentCol].ObjectType ==
Empty))
                    Ship.CurrentRow--;
                else
                    { Ship.CurrentRow--; Ship.CurrentCol--; }
            }
            else {
                if ((Scan(Astern)==false) && (Ship.CurrentRow>0))
                    Ship.CurrentRow--;
                else
                    Ship.Clockwise = !Ship.Clockwise;
            }
        }
    }
END;

```

```

Go_AntiClockwise: BEGIN
    if (Scan(TopRight)==false) {
        if ((Minefield[Ship.CurrentRow+1][Ship.CurrentCol+1].BeenHere > 1)
&&
            (Minefield[Ship.CurrentRow-1][Ship.CurrentCol].ObjectType ==
Empty))
            Ship.CurrentRow--;
    }

```



```

else
    { Ship.CurrentRow++; Ship.CurrentCol++; }
}
else {
    if (Scan(Right)==false) {
        if ((Minefield[Ship.CurrentRow][Ship.CurrentCol+1].BeenHere > 1) &&
            (Minefield[Ship.CurrentRow-1][Ship.CurrentCol].ObjectType ==
Empty))
            Ship.CurrentRow--;
        else
            Ship.CurrentCol++;
    }
    else {
        if (Scan(BottomRight)==false) {
            if ((Minefield[Ship.CurrentRow-1][Ship.CurrentCol+1].BeenHere > 1)
&&
            (Minefield[Ship.CurrentRow-1][Ship.CurrentCol].ObjectType ==
Empty))
                Ship.CurrentRow--;
            else
                { Ship.CurrentRow--; Ship.CurrentCol++; }
        }
        else {
            if ((Scan(Astern)==false) && (Ship.CurrentRow>0))
                Ship.CurrentRow--;
            else
                Ship.Clockwise = !Ship.Clockwise;
        }
    }
}
END;
}

```

B. THE MINE AVOIDANCE SIMULATION METALS INTERMEDIATE PARSED TREE

```

<.Title:<.Title:'Mine_Avoidance',Description:(.'Version' '0' '!'
    5 'A' '.').>,
Headers:(.'Settings' .),
Worlds:
(
  <.World_Name:'Minefield',
    MaxRows:32,
    MaxCols:32,
    Terrains:
    (.'Empty'
      'Nombo' 'NonMine' 'False_Alarm' 'Mine' 'Explosion' 'Boundary'
      'Start' 'Finish'
    .)
  ,
  Attributes:(.'/' 'Additional' 'Attribs' 'Can' 'Be' 'Added'
    'Here' .)
  .>
.)
,
Entities:
(
  <.Entity_Name:'Simulation',
    Entity_Attributes:(.'long'
      'MaxRuns' ',' 'numClockwise' ',' 'numAntiClockwise' ';' .)
  .> <.Entity_Name:'Run',Entity_Attributes:(.'long'
    'MaxSteps' ';' .).>
  <.Entity_Name:'Ship',

```

```

Entity_Attributes:
('bool'
 'Clockwise' ',' 'int' 'CurrentCol' ',' 'CurrentRow' ',' 'StartCol'
 ',' 'FinishCol' ','
.)

.>
.)
,
Events:
(<.Event_Name:'Initialize'.> <.Event_Name:'Move'.>
 <.Event_Name:'Go_Clockwise',
  Event_Attributes:<.Var:'DET',Mean:6,Duration:6,Additional_Attribs:'None'.>
.>

<.Event_Name:'Go_AntiClockwise',
  Event_Attributes:<.Var:'DET',Mean:6,Duration:6,Additional_Attribs:'None'.>
.>
<.Event_Name:'Start_Simulation'.> <.Event_Name:'Start_Run'.>
<.Event_Name:'Populate'.>

.)
,
Chains:
(
<.Chain_Name:'Algorithm1',
  Rules:
  (
    <.Rule_Name:'Start_Simulation',
    Patterns:
    (

```

```

<.Type:'Action',
  Body:
    (.Simulation
      '! 'MaxRuns' '=' 100 ';' 'Simulation' '! 'numClockwise'
      '=' '0' ';' 'Simulation' '! 'numAntiClockwise' '='
      '0' ';'
    .)

```

```

.>

```

```

<.Type:'Iteration',
  Body:
    <.Repeat:( '=' (.Simulation
      '! 'MaxRuns' .) .),
  This:
    (
      <.Type:'Simple',
      Body:<.Event_Name:'Start_Run',order:0.>
    .>

```

```

<.Type:'Action',
  Body:
    (.Start_Simulation' '! 'Duration'
      '+=' 'Start_Run' '! 'Duration' ';'
    .)

```

```

.>

```

```

.)

```

```

.>

```

```

.>

```

```

<.Type:'Action',

```

Body:

```
('Draw' '(' ')' ';' 'double'
'Avg_Duration' '=' 'Start_Simulation' '.' 'Duration'
 '/' 'Simulation' '.' 'MaxRuns' ';' 'cout' '<<' 'Average duration per run over '
 '<<' 'Simulation' '.' 'MaxRuns' '<<' ' run(s) is '
 '<<' 'Avg_Duration' '<<' '\n' ';' 'cout' '<<' 'Total number of events occurred = '
 '<<' 'Global_Event_Count' '<<' '\n' ';' 'cout' '<<'
'Number of Clockwise Movements = ' '<<' 'Simulation' '.'
'numClockwise' '<<' '\n' ';' 'cout' '<<' 'Number of AntiClockwise Movements
```

= '

```
'<<' 'Simulation' '.' 'numAntiClockwise' '<<' '\n'
';'
```

.)

.>

.)

,

```
Used_Event_Names:<.Start_Run:'T'.>,
number_of_occurrences:<.Start_Run:1.>
```

.>

```
<.Rule_Name:'Start_Run',
```

Patterns:

```
(<.Type:'Simple',Body:<.Event_Name:'Initialize',order:0.>.>
```

```
<.Type:'Simple',Body:<.Event_Name:'Populate',order:0.>.>
```

```
<.Type:'Action',
```

Body:

```
('long'
```

```
'Steps' '=' '0' ';' 'while' '(' '(' 'Ship' '.'
```

```
'CurrentRow' '<=' 'Minefield_MaxRows' '-' 2 ')'
```

```
'&&' '(' 'Steps' '<' 'Run' '.' 'MaxSteps' ')')'
```

```

        '{ 'Move_class' 'Move' '; 'Move' '=' 'Move_rule'
        '(' 'Move' ')' '; 'Start_Run' '.' 'Duration' '+='
        'Move' '.' 'Duration' '; '/' 'Draw' '(' ')' ';
        'Steps' '++' '; }' '/' 'Draw' '(' ')' ';

    .)

.>
.)
,
Used_Event_Names:<.Initialize:'T',Populate:'T'.>,
number_of_occurrences:<.Initialize:1,Populate:1.>
.>

<.Rule_Name:'Initialize',
Patterns:
(
    <.Type:'Action',
    Body:
    (.Run'
        '! 'MaxSteps' '=' 200 '; 'NumSquares' '=' '('
        'Minefield_MaxCols' '-' 2 ') '*' '(' 'Minefield_MaxRows'
        '-' 2 ')'; 'for' '(' 'int' 'Row' '=' '0'
        '; 'Row' '<=' 'Minefield_MaxRows' '-' 1 ';
        'Row' '++' ')' 'for' '(' 'int' 'Col' '=' '0'
        '; 'Col' '<=' 'Minefield_MaxCols' '-' 1 ';
        'Col' '++' ')' '{ 'Minefield' '[' 'Row' ']' '['
        'Col' ']' '.' 'Marked' '=' 'false' '; 'Minefield'
        '[' 'Row' ']' '[' 'Col' ']' '.' 'BeenHere' '='
        '0' '; 'Minefield' '[' 'Row' ']' '[' 'Col' ']'
        '! 'ObjectType' '=' 'Empty' '; }' 'int' 'Entry_Gap'

```

```

    [' 2 '] '=' '{ 5 ', 25 }' ';
    'int' 'Exit_Gap' [' 2 '] '=' '{ 5 ',
9 }' '; 'for' '(' 'int' 'Col' '=' '0' ';
'Col' '<=' 'Minefield_MaxCols' '-' 1 '; 'Col'
'++' ')' '{ 'if' '(' '(' 'Col' '<' 'Entry_Gap'
[' 0' ']' ')' '||' '(' 'Col' '>' 'Entry_Gap'
[' 1' ']' ')' ')' 'Minefield' [' 0' ']'
[' 'Col' ']' '.' 'ObjectType' '=' 'Boundary' ';
'else' 'Minefield' [' 0' ']' [' 'Col' ']' '.'
'ObjectType' '=' 'Empty' '; 'if' '(' '(' 'Col'
'<' 'Exit_Gap' [' 0' ']' ')' '||' '(' 'Col'
'>' 'Exit_Gap' [' 1' ']' ')' ')' 'Minefield'
[' 'Minefield_MaxRows' '-' 1 ']' [' 'Col' ']'
'.' 'ObjectType' '=' 'Boundary' '; 'else' 'Minefield'
[' 'Minefield_MaxRows' '-' 1 ']' [' 'Col' ']'
'.' 'ObjectType' '=' 'Empty' '; 'if' '(' '(' 'Col'
'==' '0' ')' '||' '(' 'Col' '==' 'Minefield_MaxCols'
'-' 1 ')' ')' '{ 'for' '(' 'int' 'Row' '='
1 '; 'Row' '<=' 'Minefield_MaxRows' '-' 2
'; 'Row' '++' ')' 'Minefield' [' 'Row' ']' ['
'Col' ']' '.' 'ObjectType' '=' 'Boundary' '; '}'
'}' 'Ship' '.' 'Clockwise' '=' 'true' '; 'srand'
('(' '(' 'unsigned' ') 'time' '(' '0' ') ')
'; 'Ship' '.' 'CurrentCol' '=' 'rand' '(' ')
'%' '(' 'Entry_Gap' [' 1' ']' '-' 'Entry_Gap'
[' 0' ']' ')' '; '/' 'Ship' '.' 'CurrentCol'
'=' 15 '; 'Ship' '.' 'CurrentRow' '=' '0' ';
'Ship' '.' 'StartCol' '=' 'Ship' '.' 'CurrentCol'
';
.)

```

```

.>
.)

.>
<.Rule_Name:'Populate',
Patterns:
(
  <.Type:'Action',
  Body:
  (.'bool'
    'IsObject' '[' 'NumSquares' ']' ',' 'IsMine' '['
    'NumSquares' ']' ',' 'IsNombo' '[' 'NumSquares' ']'
    ';' 'int' 'Objects_Per_Square_km' '=' 5 ';' 'int'
    'Mines_Per_Square_km' '=' 3 ';' 'int' 'Size_Of_Square'
    '=' 200 ';' 'int' 'Num_Squares_Per_Square_km' '='
    '(' 1000 '/' 'Size_Of_Square' ')' '*' '(' 1000
    '/' 'Size_Of_Square' ')' ';' 'int' 'ExpectedNumMines'
    '=' '(' 'NumSquares' '/' 'Num_Squares_Per_Square_km'
    ')' '*' 'Mines_Per_Square_km' ';' 'int' 'ExpectedNumObjects'
    '=' '(' 'NumSquares' '/' 'Num_Squares_Per_Square_km'
    ')' '*' 'Objects_Per_Square_km' ';' 'int' 'x' ';'
    'for' '(' 'x' '=' 0 ';' 'x' <= 'NumSquares'
    '-' 1 ';' 'x' '++' ')' '{ 'IsMine' '[' 'x'
    ']' '=' 'false' ';' 'IsObject' '[' 'x' ']' '='
    'false' ';' 'IsNombo' '[' 'x' ']' '=' 'false' ';'
    '}' 'int' 'i' ',' 'j' ',' 'Choice' ';' 'srand'
    '(' '(' 'unsigned' ')' 'time' '(' 0 ')' ')'
    ';' 'for' '(' 'i' '=' 0 ';' 'i' <= 'ExpectedNumObjects'
    '-' 1 ';' 'i' '++' ')' '{ 'Choice' '=' 1
    '+' 'rand' '(' ')' '%' 'NumSquares' ';' 'IsObject'
    '[' 'Choice' '-' 1 ']' '=' 'true' ';' '}'
  )
)

```



```

'for' ('j' '=' '0' ';' 'j' '<=' 'ExpectedNumMines'
'- 1' ';' 'j' '++') '{ 'Choice' '=' 1
'+' 'rand' '(' ')' '%' 'NumSquares' ';' 'if' '('
'IsObject' '[' 'Choice' '-' 1 ']' '==' 'false'
') 'IsMine' '[' 'Choice' '-' 1 ']' '=' 'true'
';' 'else' '{ 'IsObject' '[' 'Choice' '-' 1
']' '==' 'false' ';' 'IsNombo' '[' 'Choice' '-'
1 ']' '=' 'true' ';' }' }' 'int' 'CurrentPointer'
'=' '0' ';' 'for' ('int' 'Row' '=' 1 ';'
'Row' '<=' 'Minefield_MaxRows' '-' 2 ';' 'Row'
'++') '{ 'for' ('int' 'Col' '=' 1
';' 'Col' '<=' 'Minefield_MaxCols' '-' 2 ';'
'Col' '++') '{ 'Minefield' '[' 'Row' ']' '['
'Col' ']' '.' 'ObjectType' '=' 'Empty' ';' 'if'
'(' 'IsMine' '[' 'CurrentPointer' ']' '==' 'true'
') 'Minefield' '[' 'Row' ']' '[' 'Col' ']' '.'
'ObjectType' '=' 'Mine' ';' 'if' '(' 'IsObject' '['
'CurrentPointer' ']' '==' 'true' ') 'Minefield' '['
'Row' ']' '[' 'Col' ']' '.' 'ObjectType' '=' 'NonMine'
';' 'if' '(' 'IsNombo' '[' 'CurrentPointer' ']' '=='
'true' ') 'Minefield' '[' 'Row' ']' '[' 'Col'
']' '.' 'ObjectType' '=' 'Nombo' ';' 'CurrentPointer'
'++' ';' }' }'
.)

```

.>

.)

.>

<.Rule_Name:'Move',

Patterns:

```

(
  <.Type:'Action',
  Body:
    ('Minefield'
      '[' 'Ship' '.' 'CurrentRow' ']' '[' 'Ship' '.'
      'CurrentCol' ']' '.' 'BeenHere' '++' ';' 'if' '('
      'Ship' '.' 'CurrentRow' '<=' 'Minefield_MaxRows' '-'
      1 ')' '{' 'if' '(' '(' 'Minefield' '[' 'Ship'
      '.' 'CurrentRow' '+' 1 ']' '[' 'Ship' '.' 'CurrentCol'
      ']' '.' 'ObjectType' '==' 'Empty' ')' '&&' '('
      'Minefield' '[' 'Ship' '.' 'CurrentRow' '+' 1
      ']' '[' 'Ship' '.' 'CurrentCol' ']' '.' 'BeenHere'
      '==' '0' ')' ')' 'Ship' '.' 'CurrentRow' '++' ';'
      'else' '{' 'if' '(' 'Ship' '.' 'CurrentCol' '=='
      1 ')' 'Ship' '.' 'Clockwise' '==' 'false' ';'
      'if' '(' 'Ship' '.' 'CurrentCol' '==' 'Minefield_MaxCols'
      '-' 2 ')' 'Ship' '.' 'Clockwise' '==' 'true'
      ';' 'if' '(' 'Ship' '.' 'Clockwise' ')' '{' 'Go_Clockwise_class'
      'Go_Clockwise' ';' 'Go_Clockwise' '==' 'Go_Clockwise_rule'
      '(' 'Go_Clockwise' ')' ';' 'Simulation' '.' 'numClockwise'
      '==' 'Go_Clockwise' '.' 'Instances' ';' 'Move' '.'
      'Duration' '==' 'Go_Clockwise' '.' 'Duration' ';' '}'
      'else' '{' 'Go_AntiClockwise_class' 'Go_AntiClockwise'
      ';' 'Go_AntiClockwise' '==' 'Go_AntiClockwise_rule' '('
      'Go_AntiClockwise' ')' ';' 'Simulation' '.' 'numAntiClockwise'
      '==' 'Go_AntiClockwise' '.' 'Instances' ';' 'Move' '.'
      'Duration' '==' 'Go_AntiClockwise' '.' 'Duration' ';'
      '}' '}' '}' 'Ship' '.' 'FinishCol' '==' 'Ship' '.'
      'CurrentCol' ';'
    )
)

```

```

.>
.)

.>
<.Rule_Name:'Go_Clockwise',
Patterns:
(
  <.Type:'Action',
  Body:
  (.if
    '(' 'Scan' '(' 'TopLeft' ') '==' 'false' ') '{'
    'if' '(' '(' 'Minefield' '[' 'Ship' '.' 'CurrentRow'
    '+' 1 ']' '[' 'Ship' '.' 'CurrentCol' '-' 1
    ']' '.' 'BeenHere' '>' 1 ') '&&' '(' 'Minefield'
    '[' 'Ship' '.' 'CurrentRow' '-' 1 ']' '[' 'Ship'
    '.' 'CurrentCol' ']' '.' 'ObjectType' '==' 'Empty'
    ')') 'Ship' '.' 'CurrentRow' '--' ';' 'else'
    '{' 'Ship' '.' 'CurrentRow' '++' ';' 'Ship' '.'
    'CurrentCol' '--' ';' '}' '}' 'else' '{' 'if' '('
    'Scan' '(' 'Left' ') '==' 'false' ') '{' 'if'
    '(' '(' 'Minefield' '[' 'Ship' '.' 'CurrentRow' ']'
    '[' 'Ship' '.' 'CurrentCol' '-' 1 ']' '.' 'BeenHere'
    '>' 1 ') '&&' '(' 'Minefield' '[' 'Ship' '.'
    'CurrentRow' '-' 1 ']' '[' 'Ship' '.' 'CurrentCol'
    ']' '.' 'ObjectType' '==' 'Empty' ')') 'Ship'
    '.' 'CurrentRow' '--' ';' 'else' 'Ship' '.' 'CurrentCol'
    '--' ';' '}' 'else' '{' 'if' '(' 'Scan' '(' 'BottomLeft'
    ') '==' 'false' ') '{' 'if' '(' '(' 'Minefield'
    '[' 'Ship' '.' 'CurrentRow' ']' '[' 'Ship' '.'
    'CurrentCol' '-' 1 ']' '.' 'BeenHere' '>' 1
    ') '&&' '(' 'Minefield' '[' 'Ship' '.' 'CurrentRow'

```

```

'-' 1 ']' '[' 'Ship' '.' 'CurrentCol' ']' '.'
'ObjectType' '==' 'Empty' ')' ')' 'Ship' '.' 'CurrentRow'
'--' ';' 'else' '{' 'Ship' '.' 'CurrentRow' '--'
';' 'Ship' '.' 'CurrentCol' '--' ';' '}' '}' 'else'
'{ 'if' '(' '(' 'Scan' '(' 'Astern' ')' '=='
'false' ')' '&&' '(' 'Ship' '.' 'CurrentRow' '>'
'0' ')' ')' 'Ship' '.' 'CurrentRow' '--' ';' 'else'
'Ship' '.' 'Clockwise' '=' '!' 'Ship' '.' 'Clockwise'
';' '}' '}' '}' '}'
.)

```

```

.>
.)

```

```

.>

```

```

<.Rule_Name:'Go_AntiClockwise',

```

```

Patterns:

```

```

(

```

```

<.Type:'Action',

```

```

Body:

```

```

(.if

```

```

'(' 'Scan' '(' 'TopRight' ')' '==' 'false' ')'
'{ 'if' '(' '(' 'Minefield' '[' 'Ship' '.' 'CurrentRow'
'+' 1 ']' '[' 'Ship' '.' 'CurrentCol' '+' 1
']' '.' 'BeenHere' '>' 1 ')' '&&' '(' 'Minefield'
 '[' 'Ship' '.' 'CurrentRow' '-' 1 ']' '[' 'Ship'
 '.' 'CurrentCol' ']' '.' 'ObjectType' '==' 'Empty'
 ')' ')' 'Ship' '.' 'CurrentRow' '--' ';' 'else'
 '{ 'Ship' '.' 'CurrentRow' '++' ';' 'Ship' '.'
 'CurrentCol' '++' ';' '}' '}' 'else' '{ 'if' '('
 'Scan' '(' 'Right' ')' '==' 'false' ')' '{ 'if'

```

```

'(' (' 'Minefield' '[' 'Ship' '.' 'CurrentRow' ']'
 '[' 'Ship' '.' 'CurrentCol' '+' 1 ']' '.' 'BeenHere'
 '>' 1 ') '&&' (' 'Minefield' '[' 'Ship' '.'
 'CurrentRow' '-' 1 ']' '[' 'Ship' '.' 'CurrentCol'
 ']' '.' 'ObjectType' '==' 'Empty' ')') 'Ship'
 '.' 'CurrentRow' '--' ';' 'else' 'Ship' '.' 'CurrentCol'
 '++' ';' '}' 'else' '{' 'if' '(' 'Scan' '(' 'BottomRight'
 ')' '==' 'false' ') '{' 'if' '(' (' 'Minefield'
 '[' 'Ship' '.' 'CurrentRow' '-' 1 ']' '[' 'Ship'
 '.' 'CurrentCol' '+' 1 ']' '.' 'BeenHere' '>'
 1 ') '&&' (' 'Minefield' '[' 'Ship' '.' 'CurrentRow'
 '-' 1 ']' '[' 'Ship' '.' 'CurrentCol' ']' '.'
 'ObjectType' '==' 'Empty' ')') 'Ship' '.' 'CurrentRow'
 '--' ';' 'else' '{' 'Ship' '.' 'CurrentRow' '--'
 ';' 'Ship' '.' 'CurrentCol' '++' ';' '}' '}' 'else'
 '{' 'if' '(' (' 'Scan' '(' 'Astern' ') '=='
 'false' ') '&&' (' 'Ship' '.' 'CurrentRow' '>'
 '0' ')') 'Ship' '.' 'CurrentRow' '--' ';' 'else'
 'Ship' '.' 'Clockwise' '=' '!' 'Ship' '.' 'Clockwise'
 ';' '}' '}' '}' '}'
.)

```

.>

.)

.>

.)

.>

.)

.>

APPENDIX C : MINE AVOIDANCE SIMULATION IN GENERATED C++

A. THE MINE AVOIDANCE SIMULATION GENERATED C++ MAIN.CPP

```
// METALS Code Generator Version 0.4C
// C++ simulation program [Mine_Avoidance] created from Mine5.txt
// Version 0 . 5 A

#include <iostream>
#include <ctime>
#include <cstdlib>
#include <stdlib.h>

using namespace std;

#include "Worlds.h"
#include "Entities.h"
#include "Settings.h"
#include "Events.h"
#include "Chains.h"

// Main
int main(int argc, char *argv[])
{
    Start_Simulation_class Start_Simulation;
    Start_Simulation_rule( Start_Simulation );

    system("PAUSE");
    return 0;
}
```

B. THE MINE AVOIDANCE SIMULATION GENERATED WORLDS.H

```
// METALS Code Generator Version 0.4C
```

```
// Generation of Worlds
```

```
int Minefield_MaxRows = 32;
```

```
int Minefield_MaxCols = 32;
```

```
enum Minefield_Terrain { Empty, Nombo, NonMine, False_Alarm, Mine, Explosion,  
Boundary, Start, Finish };
```

```
class Minefield_class
```

```
{
```

```
    public:
```

```
        Minefield_class();
```

```
        int BeenHere;
```

```
        bool Marked;
```

```
        Minefield_Terrain ObjectType;
```

```
        // Additional Attribs Can Be Added Here
```

```
};
```

```
Minefield_class :: Minefield_class()
```

```
{
```

```
    BeenHere = 0;
```

```
    Marked = false;
```

```
    ObjectType = Empty;
```

```
}
```

```
Minefield_class Minefield[32][32];
```

C. THE MINE AVOIDANCE SIMULATION GENERATED ENTITIES.H

// METALS Code Generator Version 0.4C

// Generation of Entities

```
class Simulation_class
{
    public:

        long MaxRuns , numClockwise , numAntiClockwise;

};

class Run_class
{
    public:

        long MaxSteps;

};

class Ship_class
{
    public:

        bool Clockwise;
        int CurrentCol , CurrentRow , StartCol , FinishCol;

};

Simulation_class Simulation;
Run_class Run;
Ship_class Ship;
```


D. THE MINE AVOIDANCE SIMULATION GENERATED EVENTS.H

```
// METALS Code Generator Version 0.4C

// Generation of Event Classes

long Global_Event_Count;

enum EType { DET, STO };

class Initialize_class
{
public:
    Initialize_class();
    string Name;
    EType Var;
    double Mean;
    double Duration;
    static long Instances;

};

long Initialize_class :: Instances = 0;

Initialize_class :: Initialize_class()
{
    Global_Event_Count++;
    Instances++;

    Name = "Initialize";
    Var = DET;
    Mean = 0;
    Duration = 0;

    if (Var==STO) // Discrete Poisson RNG.
    {
        double a = exp(-Mean);
        double p = 1;
        long x = 1;
        while (p > a) {
            double U = (double) rand() / RAND_MAX;
            p = p*U;
            x++; }
        Duration = x; }
}
```

```

class Move_class
{
public:
    Move_class();
    string Name;
    EType Var;
    double Mean;
    double Duration;
    static long Instances;

};

long Move_class :: Instances = 0;

Move_class :: Move_class()
{
    Global_Event_Count++;
    Instances++;

    Name = "Move";
    Var = DET;
    Mean = 0;
    Duration = 0;

    if (Var==STO) // Discrete Poisson RNG.
    {
        double a = exp(-Mean);
        double p = 1;
        long x = 1;
        while (p > a) {
            double U = (double) rand() / RAND_MAX;
            p = p*U;
            x++; }
        Duration = x; }
    }

class Go_Clockwise_class
{
public:
    Go_Clockwise_class();
    string Name;
    EType Var;
    double Mean;
    double Duration;

```

```

        static long Instances;

};

long Go_Clockwise_class :: Instances = 0;

Go_Clockwise_class :: Go_Clockwise_class()
{
    Global_Event_Count++;
    Instances++;

    Name = "Go_Clockwise";
    Var = DET ;
        Mean = 6 ;
        Duration = 6 ;

    if (Var==STO) // Discrete Poisson RNG.
    {
        double a = exp(-Mean);
        double p = 1;
        long x = 1;
        while (p > a) {
            double U = (double) rand() / RAND_MAX;
            p = p*U;
            x++; }
        Duration = x; }
    }

class Go_AntiClockwise_class
{
public:
    Go_AntiClockwise_class();
    string Name;
    EType Var;
    double Mean;
    double Duration;
    static long Instances;

};

long Go_AntiClockwise_class :: Instances = 0;

Go_AntiClockwise_class :: Go_AntiClockwise_class()
{

```

```

Global_Event_Count++;
Instances++;

Name = "Go_AntiClockwise";
Var = DET ;
    Mean = 6 ;
    Duration = 6 ;

if (Var==STO) // Discrete Poisson RNG.
{
    double a = exp(-Mean);
    double p = 1;
    long x = 1;
    while (p > a) {
        double U = (double) rand() / RAND_MAX;
        p = p*U;
        x++; }
    Duration = x; }
}

class Start_Simulation_class
{
public:
    Start_Simulation_class();
    string Name;
    EType Var;
    double Mean;
    double Duration;
    static long Instances;

};

long Start_Simulation_class :: Instances = 0;

Start_Simulation_class :: Start_Simulation_class()
{
    Global_Event_Count++;
    Instances++;

    Name = "Start_Simulation";
    Var = DET;
    Mean = 0;
    Duration = 0;

    if (Var==STO) // Discrete Poisson RNG.

```

```

{
    double a = exp(-Mean);
    double p = 1;
    long x = 1;
    while (p > a) {
        double U = (double) rand() / RAND_MAX;
        p = p*U;
        x++; }
    Duration = x; }
}

class Start_Run_class
{
public:
    Start_Run_class();
    string Name;
    EType Var;
    double Mean;
    double Duration;
    static long Instances;

};

long Start_Run_class :: Instances = 0;

Start_Run_class :: Start_Run_class()
{
    Global_Event_Count++;
    Instances++;

    Name = "Start_Run";
    Var = DET;
    Mean = 0;
    Duration = 0;

    if (Var==STO) // Discrete Poisson RNG.
    {
        double a = exp(-Mean);
        double p = 1;
        long x = 1;
        while (p > a) {
            double U = (double) rand() / RAND_MAX;
            p = p*U;
            x++; }
        Duration = x; }
}

```

```

    }

class Populate_class
{
public:
    Populate_class();
    string Name;
    EType Var;
    double Mean;
    double Duration;
    static long Instances;

};

long Populate_class :: Instances = 0;

Populate_class :: Populate_class()
{
    Global_Event_Count++;
    Instances++;

    Name = "Populate";
    Var = DET;
    Mean = 0;
    Duration = 0;

    if (Var==STO) // Discrete Poisson RNG.
    {
        double a = exp(-Mean);
        double p = 1;
        long x = 1;
        while (p > a) {
            double U = (double) rand() / RAND_MAX;
            p = p*U;
            x++; }
        Duration = x; }
    }

```

E. THE MINE AVOIDANCE SIMULATION GENERATED CHAINS.H

```
// METALS Code Generator Version 0.4C

// Generation of Event Chains

// The various user defined event descriptions in Algorithm1

// Functional prototypes for each rule in the event chain
Start_Simulation_class Start_Simulation_rule( Start_Simulation_class );
Start_Run_class Start_Run_rule( Start_Run_class );
Initialize_class Initialize_rule( Initialize_class );
Populate_class Populate_rule( Populate_class );
Move_class Move_rule( Move_class );
Go_Clockwise_class Go_Clockwise_rule( Go_Clockwise_class );
Go_AntiClockwise_class Go_AntiClockwise_rule( Go_AntiClockwise_class );

Start_Simulation_class Start_Simulation_rule( Start_Simulation_class )
{
    Start_Simulation_class Start_Simulation;

Start_Run_class Start_Run; Simulation . MaxRuns = 100;
    Simulation . numClockwise = 0;
    Simulation . numAntiClockwise = 0;

for ( __int64 i=1; i<=Simulation.MaxRuns; i++)
{
    Start_Run =Start_Run_rule (Start_Run); Start_Simulation . Duration += Start_Run .
Duration;

} Draw ();
    double Avg_Duration = Start_Simulation . Duration / Simulation . MaxRuns;
    cout <<"Average duration per run over "
    << Simulation . MaxRuns <<" run(s) is "
    << Avg_Duration <<"\n"
    ;
    cout <<"Total number of events occurred = "
    << Global_Event_Count <<"\n"
    ;
    cout <<"Number of Clockwise Movements = "
    << Simulation . numClockwise <<"\n"
    ;
    cout <<"Number of AntiClockwise Movements = "
    << Simulation . numAntiClockwise <<"\n"
```

```
;
```

```
    return Start_Simulation;  
}
```

```
Start_Run_class Start_Run_rule( Start_Run_class )  
{  
    Start_Run_class Start_Run;
```

```
    Initialize_class Initialize;  
    Populate_class Populate;  
    Initialize = Initialize_rule (Initialize);  
    Populate = Populate_rule (Populate); long Steps = 0;  
    while ( ( Ship . CurrentRow <= Minefield_MaxRows - 2 ) && ( Steps < Run .  
MaxSteps ) ) { Move_class Move;  
        Move = Move_rule ( Move );  
        Start_Run . Duration += Move . Duration;  
        // Draw ( );  
        Steps ++;  
    } // Draw ( );
```

```
    return Start_Run;  
}
```

```
Initialize_class Initialize_rule( Initialize_class )  
{  
    Initialize_class Initialize;  
    Run . MaxSteps = 200;  
    NumSquares = ( Minefield_MaxCols - 2 ) * ( Minefield_MaxRows - 2 );  
    for ( int Row = 0;  
        Row <= Minefield_MaxRows - 1;  
        Row ++ ) for ( int Col = 0;  
            Col <= Minefield_MaxCols - 1;  
            Col ++ ) { Minefield [ Row ] [ Col ] . Marked = false;  
                Minefield [ Row ] [ Col ] . BeenHere = 0;  
                Minefield [ Row ] [ Col ] . ObjectType = Empty;  
            } int Entry_Gap [ 2 ] = { 5 , 25 };  
            int Exit_Gap [ 2 ] = { 5 , 9 };  
            for ( int Col = 0;  
                Col <= Minefield_MaxCols - 1;  
                Col ++ ) { if ( ( Col < Entry_Gap [ 0 ] ) || ( Col > Entry_Gap [ 1 ] ) ) Minefield [ 0 ] [ Col ] . ObjectType = Boundary;
```



```

        else Minefield [ 0 ] [ Col ] . ObjectType = Empty;
        if ( ( Col < Exit_Gap [ 0 ] ) || ( Col > Exit_Gap [ 1 ] ) ) Minefield [
Minefield_MaxRows - 1 ] [ Col ] . ObjectType = Boundary;
        else Minefield [ Minefield_MaxRows - 1 ] [ Col ] . ObjectType = Empty;
        if ( ( Col == 0 ) || ( Col == Minefield_MaxCols - 1 ) ) { for ( int Row = 1;
Row <= Minefield_MaxRows - 2;
Row ++ ) Minefield [ Row ] [ Col ] . ObjectType = Boundary;
        } } Ship . Clockwise = true;
        srand ( ( unsigned ) time ( 0 ) );
        Ship . CurrentCol = rand ( ) % ( Entry_Gap [ 1 ] - Entry_Gap [ 0 ] );
        // Ship . CurrentCol = 15;
        Ship . CurrentRow = 0;
        Ship . StartCol = Ship . CurrentCol;

    return Initialize;
}

```

```

Populate_class Populate_rule( Populate_class )
{
    Populate_class Populate;
    bool IsObject [ NumSquares ] , IsMine [ NumSquares ] , IsNombo [ NumSquares ];
    int Objects_Per_Square_km = 5;
    int Mines_Per_Square_km = 3;
    int Size_Of_Square = 200;
    int Num_Squares_Per_Square_km = ( 1000 / Size_Of_Square ) * ( 1000 /
Size_Of_Square );
    int ExpectedNumMines = ( NumSquares / Num_Squares_Per_Square_km ) *
Mines_Per_Square_km;
    int ExpectedNumObjects = ( NumSquares / Num_Squares_Per_Square_km ) *
Objects_Per_Square_km;
    int x;
    for ( x = 0;
x <= NumSquares - 1;
x ++ ) { IsMine [ x ] = false;
IsObject [ x ] = false;
IsNombo [ x ] = false;
    } int i , j , Choice;
    srand ( ( unsigned ) time ( 0 ) );
    for ( i = 0;
i <= ExpectedNumObjects - 1;
i ++ ) { Choice = 1 + rand ( ) % NumSquares;
IsObject [ Choice - 1 ] = true;
    } for ( j = 0;
j <= ExpectedNumMines - 1;

```

```

j ++ ) { Choice = 1 + rand ( ) % NumSquares;
if ( IsObject [ Choice - 1 ] == false ) IsMine [ Choice - 1 ] = true;
else { IsObject [ Choice - 1 ] == false;
IsNombo [ Choice - 1 ] = true;
} } int CurrentPointer = 0;
for ( int Row = 1;
Row <= Minefield_MaxRows - 2;
Row ++ ) { for ( int Col = 1;
Col <= Minefield_MaxCols - 2;
Col ++ ) { Minefield [ Row ] [ Col ] . ObjectType = Empty;
if ( IsMine [ CurrentPointer ] == true ) Minefield [ Row ] [ Col ] . ObjectType = Mine;
if ( IsObject [ CurrentPointer ] == true ) Minefield [ Row ] [ Col ] . ObjectType =
NonMine;
if ( IsNombo [ CurrentPointer ] == true ) Minefield [ Row ] [ Col ] . ObjectType =
Nombo;
CurrentPointer ++;
} }

return Populate;
}

```

```

Move_class Move_rule( Move_class )
{
Move_class Move;
Minefield [ Ship . CurrentRow ] [ Ship . CurrentCol ] . BeenHere ++;
if ( Ship . CurrentRow <= Minefield_MaxRows - 1 ) { if ( ( Minefield [ Ship .
CurrentRow + 1 ] [ Ship . CurrentCol ] . ObjectType == Empty ) && ( Minefield [ Ship .
CurrentRow + 1 ] [ Ship . CurrentCol ] . BeenHere == 0 ) ) Ship . CurrentRow ++;
else { if ( Ship . CurrentCol == 1 ) Ship . Clockwise = false;
if ( Ship . CurrentCol == Minefield_MaxCols - 2 ) Ship . Clockwise = true;
if ( Ship . Clockwise ) { Go_Clockwise_class Go_Clockwise;
Go_Clockwise = Go_Clockwise_rule ( Go_Clockwise );
Simulation . numClockwise = Go_Clockwise . Instances;
Move . Duration = Go_Clockwise . Duration;
} else { Go_AntiClockwise_class Go_AntiClockwise;
Go_AntiClockwise = Go_AntiClockwise_rule ( Go_AntiClockwise );
Simulation . numAntiClockwise = Go_AntiClockwise . Instances;
Move . Duration = Go_AntiClockwise . Duration;
} } } Ship . FinishCol = Ship . CurrentCol;

return Move;
}

```

```

Go_Clockwise_class Go_Clockwise_rule( Go_Clockwise_class )
{
    Go_Clockwise_class Go_Clockwise;
    if( Scan ( TopLeft ) == false ) { if ( ( Minefield [ Ship . CurrentRow + 1 ] [ Ship .
CurrentCol - 1 ] . BeenHere > 1 ) && ( Minefield [ Ship . CurrentRow - 1 ] [ Ship .
CurrentCol ] . ObjectType == Empty ) ) Ship . CurrentRow --;
        else { Ship . CurrentRow ++;
            Ship . CurrentCol --;
        } } else { if( Scan ( Left ) == false ) { if ( ( Minefield [ Ship . CurrentRow ] [ Ship .
CurrentCol - 1 ] . BeenHere > 1 ) && ( Minefield [ Ship . CurrentRow - 1 ] [ Ship .
CurrentCol ] . ObjectType == Empty ) ) Ship . CurrentRow --;
            else Ship . CurrentCol --;
        } } else { if( Scan ( BottomLeft ) == false ) { if ( ( Minefield [ Ship . CurrentRow ] [
Ship . CurrentCol - 1 ] . BeenHere > 1 ) && ( Minefield [ Ship . CurrentRow - 1 ] [ Ship .
CurrentCol ] . ObjectType == Empty ) ) Ship . CurrentRow --;
            else { Ship . CurrentRow --;
                Ship . CurrentCol --;
            } } } else { if ( ( Scan ( Astern ) == false ) && ( Ship . CurrentRow > 0 ) ) Ship .
CurrentRow --;
            else Ship . Clockwise = ! Ship . Clockwise;
        } } }

    return Go_Clockwise;
}

```

```

Go_AntiClockwise_class Go_AntiClockwise_rule( Go_AntiClockwise_class )
{
    Go_AntiClockwise_class Go_AntiClockwise;
    if( Scan ( TopRight ) == false ) { if ( ( Minefield [ Ship . CurrentRow + 1 ] [ Ship .
CurrentCol + 1 ] . BeenHere > 1 ) && ( Minefield [ Ship . CurrentRow - 1 ] [ Ship .
CurrentCol ] . ObjectType == Empty ) ) Ship . CurrentRow --;
        else { Ship . CurrentRow ++;
            Ship . CurrentCol ++;
        } } else { if( Scan ( Right ) == false ) { if ( ( Minefield [ Ship . CurrentRow ] [ Ship .
CurrentCol + 1 ] . BeenHere > 1 ) && ( Minefield [ Ship . CurrentRow - 1 ] [ Ship .
CurrentCol ] . ObjectType == Empty ) ) Ship . CurrentRow --;
            else Ship . CurrentCol ++;
        } } else { if( Scan ( BottomRight ) == false ) { if ( ( Minefield [ Ship . CurrentRow - 1 ] [
Ship . CurrentCol + 1 ] . BeenHere > 1 ) && ( Minefield [ Ship . CurrentRow - 1 ] [ Ship .
CurrentCol ] . ObjectType == Empty ) ) Ship . CurrentRow --;
            else { Ship . CurrentRow --;
                Ship . CurrentCol ++;
            } } } else { if ( ( Scan ( Astern ) == false ) && ( Ship . CurrentRow > 0 ) ) Ship .
CurrentRow --;
            else Ship . Clockwise = ! Ship . Clockwise;
        } } }

```

```
    } } }  
    return Go_AntiClockwise;  
}
```

F. THE MINE AVOIDANCE SIMULATION USER INPUT SETTINGS.H

```
// Detection Algorithm Paramters.
const double Prob_Detection = 1;
const double Prob_False_Alarm = 1;
int NumSquares;

enum ScanDirection { Ahead, TopLeft, TopRight, Left, Right,
                    BottomLeft, BottomRight, Astern };

bool Scan (ScanDirection Direction)
{
    bool Detected = false;

    int NextRow; int NextCol;
    switch (Direction)
    {
        case Ahead:    NextRow = Ship.CurrentRow+1; NextCol = Ship.CurrentCol; break;
        case TopLeft:  NextRow = Ship.CurrentRow+1; NextCol = Ship.CurrentCol-1; break;
        case Left:     NextRow = Ship.CurrentRow;   NextCol = Ship.CurrentCol-1; break;
        case BottomLeft: NextRow = Ship.CurrentRow-1; NextCol = Ship.CurrentCol-1; break;
        case Astern:   NextRow = Ship.CurrentRow-1; NextCol = Ship.CurrentCol; break;
        case BottomRight: NextRow = Ship.CurrentRow-1; NextCol = Ship.CurrentCol+1;
break;
        case Right:    NextRow = Ship.CurrentRow;   NextCol = Ship.CurrentCol+1; break;
        case TopRight:  NextRow = Ship.CurrentRow+1; NextCol = Ship.CurrentCol+1; break;
        default:       NextRow = Ship.CurrentRow;   NextCol = Ship.CurrentCol; break;
    }

    srand((unsigned)time(0));
    double Chance = double(rand()%100)/100;

    if (Minefield[NextRow][NextCol].ObjectType == Empty)
    {
        // if (Chance > Prob_Detection*Prob_False_Alarm)
        Detected = false;
        // else
        // {
        //     Minefield[NextRow][NextCol].ObjectType = False_Alarm;
        // }
    }
    else
    {
        // if (Chance <= Prob_Detection)
        Detected = true;
        // else
    }
}
```

```

    // {
    //   Minefield[NextRow][NextCol].ObjectType = Explosion;
    //   Detected = false;
    //   Sunk = true;
    // }
  }
  if (Detected) Minefield[NextRow][NextCol].Marked = true;
  return Detected;
}

void Draw()
{
  system("CLS");

  cout << "+";
  for (int Col=1; Col<=Minefield_MaxCols-2; Col++)
  {
    if (Col==Ship.FinishCol) cout << "F";
    else
      if (Minefield[Minefield_MaxRows-1][Col].BeenHere > 0)
        cout << Minefield[Minefield_MaxRows-1][Col].BeenHere;
      else
        if (Minefield[Minefield_MaxRows-1][Col].ObjectType == Boundary) cout << "-";
        else cout << " ";
  }
  cout << "+\n";

  for (int Row=Minefield_MaxRows-2; Row>=1; Row--)
  {
    for (int Col=0; Col<=Minefield_MaxCols-1; Col++)
    {
      if (Minefield[Row][Col].BeenHere > 0)
        cout << Minefield[Row][Col].BeenHere;
      else
      {
        switch (Minefield[Row][Col].ObjectType)
        {
          case Boundary: cout << "|"; break;
          case Mine: cout << "*"; break;
          case NonMine: cout << "o"; break;
          case Nombo: cout << "?"; break;
          case Explosion: cout << "K"; break;
          case False_Alarm: cout << "!"; break;
          default: cout << " ";
        }
      }
    }
  }
}

```

```

        }
    }

}
cout << "\n";
}

cout << "+";
for (int Col=1; Col<=Minefield_MaxCols-2; Col++)
{
    if (Col==Ship.StartCol) cout << "S";
    else
        if (Minefield[0][Col].BeenHere > 0)
            cout << Minefield[0][Col].BeenHere;
        else
            if (Minefield[0][Col].ObjectType == Boundary) cout << "-";
            else cout << " ";
    }
    cout << "+\n";
    cout << "\n\n";
    cout << "Current Row = " << Ship.CurrentRow << "\n";
    system( "PAUSE");
}

```

LIST OF REFERENCES

- [BALL99] Ball T, The Essence of Dynamic Analysis, Proceedings at the University of Washington/Microsoft Research Summer Institute on Technologies to Improve Software Development, UW Campus and Semiahmoo Resort August 2-6, 1999, presentation slides (also on the Microsoft web site at <http://research.microsoft.com/tisd/Slides/TomBall.ppt>), November 18, 2004.
- [LARUS95] Larus J. R., Schnarr E., *EEL: Machine-Independent Executable Editing*, Proceeding of PLDI 1995, SIGPLAN Notices, pp291-300, Vol 30, No 6, June 1995.
- [AUG98] Auguston M., *Building Program Behavior Models*, European Conference on Artificial Intelligence ECAI'98 Workshop on Temporal and Spatial Reasoning, 1998.
- [AUG03] Auguston M., Jefferey C., Underwood S., *A Monitoring Language for Run Time and Post-Mortem Behavior Analysis and Visualization*, Proceedings of 5th International Workshop on Algorithmic and Automatic Debugging AADEBUG 2003, Ghent, Belgium, pp. 41-54 (also on the CoRR web site at <http://arxiv.org/abs/cs/0310025>), September 8-10, 2003
- [NYGAARD] Dahl O. J., Nygaard K., *SIMULA: an ALGOL-based simulation language*, Communications of the ACM, Volume 9 Issue 9, pp. 671-678, September 1966.
- [BOLIER] Bolier D., Eliens A., *SIM : a C++ library for Discrete Event Simulation*, Vrije Universiteit, Department of Mathematics and Computer Science, Amsterdam, The Netherlands web site at http://www.cs.vu.nl/~eliens/sim/sim_html/sim.html, November 18, 2004.
- [CSSL67] Strauss J. C., Augustin D.C., Fineberg M.S., Johnson B.B., Linebarger R. M., Sansom F. J., *The SCI Continuous System Simulation Language (CSSL)*, pp281-303, Simulation 9(12), December 1967.
- [COMDIC1] Facts obtained from <http://computing-dictionary.thefreedictionary.com>, November 18, 2004.
- [SPECK76] Speckhart F. H. et al, *A Guide to Using CSMP - The Continuous System Modeling Program*, P-H, 1976.
- [IBM79] IBM Corporation, *IBM System/370 APL Continuous System Modelling Program Program Description and Operations Manual*, Moline (Ill.), SH20-2115, 1979.

- [ALFON99] Alfonseca M., Alfonseca E., Lara J.D., *Compiling a simulation language in APL*, Proceedings of the APL98 conference on Array Processing Language, Rome, Italy, pp. 105-109, 1999.
- [EAS-E] Facts obtained from <http://www.eas-e.org/index.html>, November 19, 2004.
- [HYPER1] Facts obtained from <http://www.hyperdictionary.com>, November 19, 2004.
- [HYPER2] Facts obtained from <http://www.hyperdictionary.com>, November 19, 2004.
- [RIG1] Auguston M., *Programming language RIGAL as a compiler writing tool*, ACM SIGPLAN Notices, vol.25, #12, pp.61-69, December 1990.
- [RIG2] Auguston M., *RIGAL - a programming language for compiler writing*, Lecture Notes in Computer Science, Springer Verlag, vol.502, pp.529-564, 1991.
- [RIG3] RIGAL Language Home Page at <http://www.ida.liu.se/~vaden/rigal/>, August 13, 2004.
- [JERRY] Jerry Banks, *Handbook of Simulation - Principles, Methodology, Advances, Applications, and Practice*, John Wiley & Sons, Inc, pp.149-166, 1998.
- [GAVER1] Donald P. Gaver, *The Effect Of Sensor Performance On Safe Minefield Transit*, Menneken Lecture 2003, 2003.
- [GAVER2] Donald P. Gaver, Patricia A. Jacobs, Steven E. Pilnick, *On Minefield Transit by Detection, Avoidance, and DeMining*, 3rd Joint Australian/American Conference on Mine Countermeasures and Demining Australian Defence Force Academy, Canberra 2004, 2004.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Mikhail Auguston
Naval Postgraduate School
Monterey, California
4. Richard Riehle
Naval Postgraduate School
Monterey, California